Commodore 64: How To Use Those Mysterious CIA Timers

The Transactor

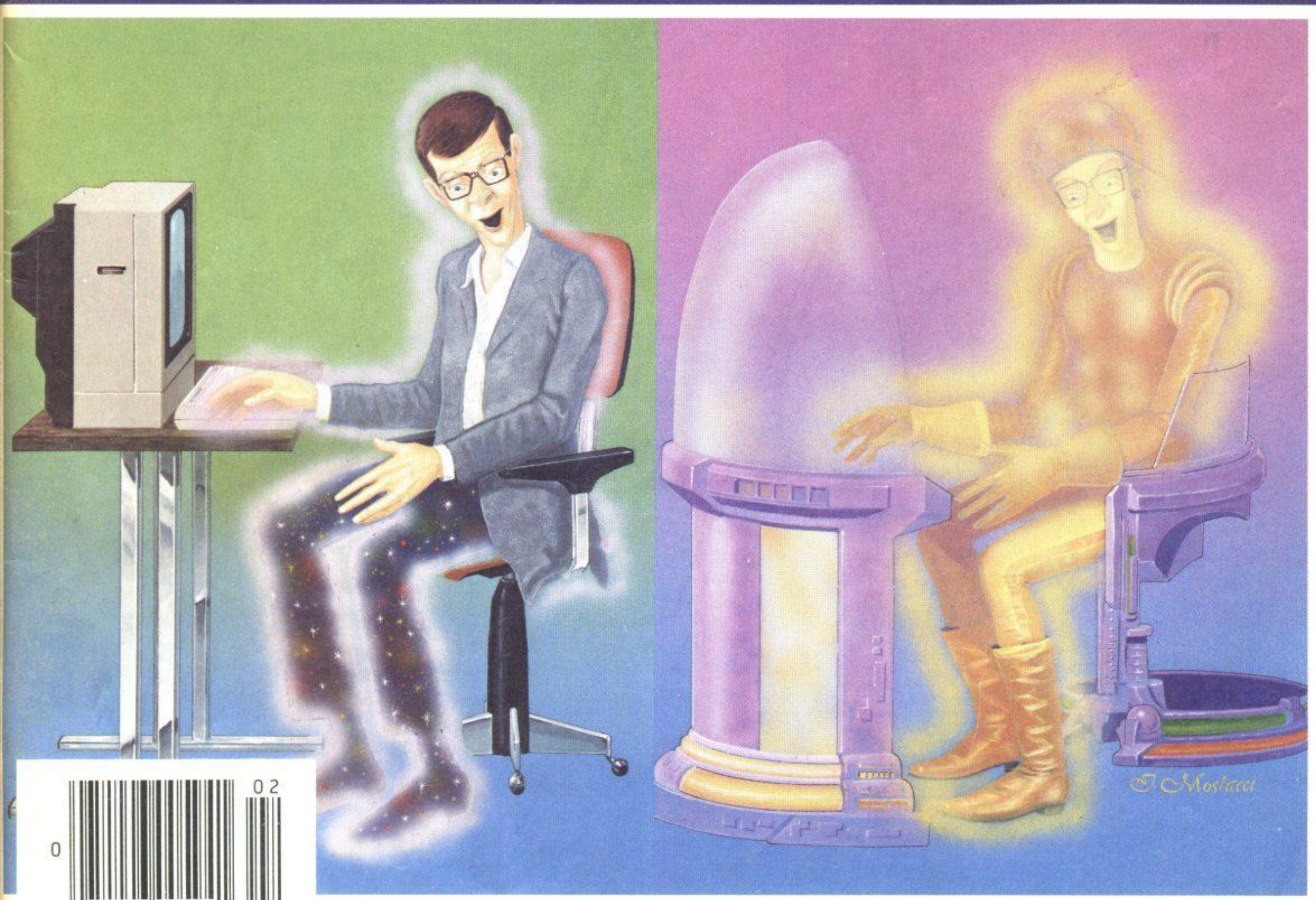
Second Class Mail Permit Pending
Postage Paid in Milton, Ontario

The Tech/News Journal For Commodore Computers Vol. 5

Issue 02
$2.95

The Transition To Machine Language

- How BASIC Works: What Happens After You Press RETURN
- Converting BASIC To Machine Language
- Rocket Thruster Simulation In BASIC AND In Machine Code
- Machine Language Monitor In BASIC: A Learning Tool
- The Stack: What Happens In This BASIC Forbidden Zone?
- Merging Commodore BASIC Programs Together
- Butterfield: Use Your Commodore 64 To Emulate The SX-64
- Plus Lots More... And ALL On Commodore!



0 72246 00594 02

The Reference Transactor Is Coming! (See News BRK for details)

INTRODUCING

THE PRO-LINE TEAM

PAL 64
The fastest and easiest to use assembler for the Commodore 64. Pal 64 enables the user to perform assembly language programming using the standard MOS mnemonics. **$69.95**

POWER 64
Is an absolutely indispensible aid to the programmer using Commodore 64 BASIC. Power 64 turbo-charges resident BASIC with dozens of new super useful commands like MERGE, UNDO, TEST and DISK as well as all the old standbys such as RENUM and SEARCH & REPLACE. Includes MorePower 64. **$69.95**

TOOL BOX 64
Is the ultimate programmer's utility package. Includes Pal 64 assembler and Power 64 BASIC soup-up kit all together in one fully integrated and economical package. **$129.95**

SPELLPRO 64
Is an easy to use spelling checker with a standard dictionary expandable to 25,000 words. SpellPro 64 quickly adapts itself to your personal vocabulary and business jargon allowing you to add and delete words to/from the dictionary, edit documents to correct unrecognized words and output lists of unrecognized words to printer or screen. SpellPro 64 was designed to work with the WordPro Series* and other wordprocessing programs using the WordPro file format. **$69.95**

NOW SHIPPING!!!

For Your Nearest Dealer
Call
(416) 273-6350

†Commodore 64 and Commodore are trademarks of Commodore Business Machines Inc.
*Presently marketed by Professional Software Inc.

Specifications subject to change without notice.

WP64
This brand new offering from the originators of the WordPro Series* brings professional wordprocessing to the Commodore 64 for the first time. Two years under development, WP64 features 100% proportional printing capability as well as 40/80 column display, automatic word wrap, two column printing, alternate paging for headers & footers, four way scrolling, extra text area and a brand new 'OOPS' buffer that magically brings back text deleted in error. All you ever dreamed of in a wordprocessor program, WP64 sets a new high standard for the software industry to meet. **$69.95**

MAILPRO 64
A new generation of data organizer and list manager, MailPro 64 is the easiest of all to learn and use. Handles up to 4,000 records on one disk, prints multiple labels across, does minor text editing ie: setting up invoices. Best of all, MailPro 64 resides entirely within memory so you don't have to constantly juggle disks like you must with other data base managers for the Commodore 64. **$69.95**

PRO·LINE SOFTWARE

(416) 273-6350
755 THE QUEENSWAY EAST, UNIT 8,
MISSISSAUGA, ONTARIO, CANADA, L4Y 4C5

CAD/CAM! DON'T SPEND 25k, 50k or $500,000 BEFORE YOU SPEND $79.00

OBJECTIVES

This book will provide managers, engineers, manufacturing personnel and any interested persons an understanding of the fundamentals of Computer Aided Design (CAD) and Computer Aided manufacturing (CAM) applications and technology.

PROGRAM DESCRIPTION

The program will expose you to the various CAD/CAM terminologies used. Hardware and software comparisons will be explored with heavy emphasis on their advantages and disadvantages. Cost justification and implementation are presented using case studies.

WHO SHOULD PARTICIPATE

The course is designed for but not limited to:

— Those managers, engineers and research professionals associated with the manufacturing industry.

— Personnel from Product, Tool Design, Plant Layout and Plant Engineering who are interested in CAD/CAM.

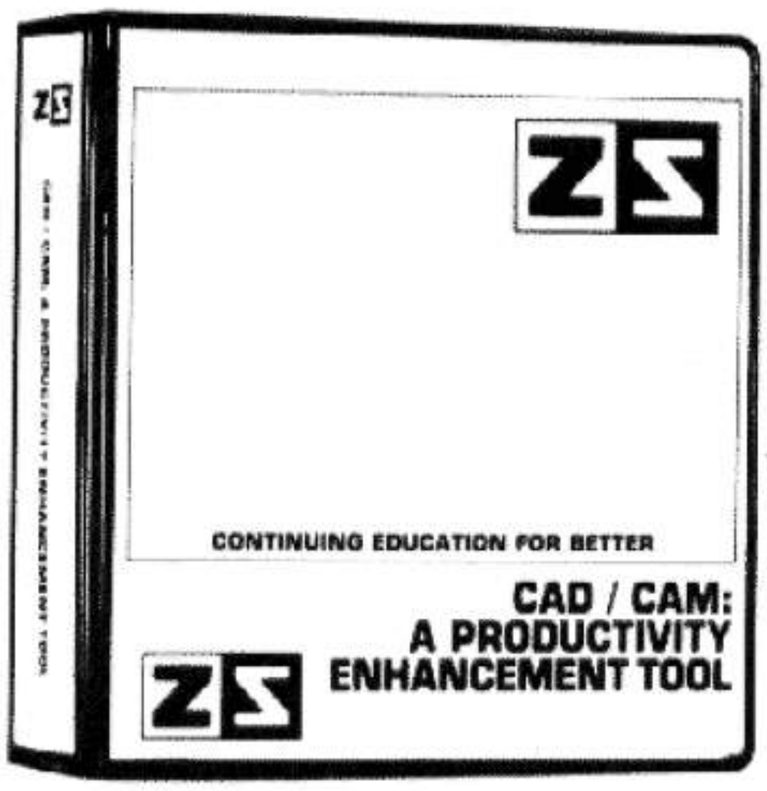
ADVANTAGES— END RESULT

This program will enable participants to:

1. Learn basic CAD/CAM Vocabulary.
2. Better understand the various hardware and software components used in a typical CAD work station.
3. Select the existing CAD/CAM system most appropriate for current and projected needs.
4. Make an effective cost justification as to Why they SHOULD or SHOULD NOT implement a CAD/CAM system.
5. Apply and use computer graphics as a productivity tool.

PROGRAM CONTENT

1. Introduction
   a. History of CAD/CAM
   b. Importance of CAD/CAM
2. Graphics work station peripherals
   a. Input
   b. Output
   c. Advantages and disadvantages of input and output devices.
3. Computer Graphics Systems (Hardware)
   a. Micros
   b. Minis
   c. Main Frames
   d. Turnkey Graphics systems
4. Software
   a. Operating systems
   b. Graphics Packages
   c. Graphics Modules
5. Computer Aided Design
   a. Geometric Definitions (Points, Lines, Circles, ETC..)
   b. Control functions
   c. Graphics Manipulations
   d. Drafting Functions
   e. Filing functions
   f. Applications
6. Implementation
   a. Determining needs
   b. Purchasing and Installing
   c. Getting Started
7. Cost Justification and Survey
   a. Cost comparisons of two and four work station systems.
   b. Presentation of recent survey of CAD system users



ZANIM SYSTEMS MAKES THIS SPECIAL OFFER: IF YOU BUY **CAD/CAM: A PRODUCTIVITY ENHANCEMENT TOOL** BEFORE APRIL 15TH, WE WILL INCLUDE *FREE OF CHARGE* THESE TWO PAPERS PUBLISHED NATIONALLY BY ZANIM SYSTEMS CAD/CAM EXPERT.

1. "Creation of a Large Data Base for a Small Graphics System"
2. "Shortest Path Algorithm Using Computer Graphics"

Of course you could spend as much as $495, $595 or $695 for a similar 3 day seminar even though this book is not a computer program.

We tell you April 15th for a special reason...this product may be tax deductible depending on your field or needs. This 170 page course will satisfy any of your CAD/CAM needs. We guarantee it.

Please send $79 to:

ZANIM SYSTEMS
CAD/CAM GROUP
P.O. BOX 4364
FLINT, MI 48504
(313) 233-5731

QUANTITY DISCOUNTS AVAILABLE FOR COLLEGES, UNIVERSITIES AND/OR SEMINAR USE.

SOFTWARE FOR
VIC ★ COMMODORE 64 ★ PET
FROM KING MICROWARE

- S D COPY FAST EFFICIENT SINGLE DISC COPIER FOR THE 1541 — $19.95
- WORDS & CALCS SPREAD SHEET FOR THE C-64 ALLOWS TEXT — $42.95
- CHART PAC 64 FINEST CHART MAKER AROUND — $42.95
- SMARTEES ACTION PACKED MAZE GAME — $22.95

NEW! THE BANKER THE FINEST CHECK BOOK RECONCILIATION PROGRAM ON THE MARKET — $38.95

- DAISY — DATA ADAPTABLE INFORMATION SYSTEM
  — THE DATA BASE WITH A DIFFERENCE
  — ALLOWS YOU TO CALCULATE BETWEEN FIELDS — $39.95
- ASTRO POSITIONS FIND THE STARS AND CAST YOUR HOROSCOPE — $43.95

LOOK AT THE LANGUAGES WE HAVE

YES! WE HAVE PASCAL — $52.95

ULTRABASIC WITH TURTLE GRAPHICS AND SOUND — $42.95

TINY BASIC COMPILER — $22.95

TINY FORTH FIG FORTH IMPLEMENTATION — $22.95

EDIT/ASM COMPLETE EDITOR ASSEMBLER PACKAGE — $36.95

64-BUDGETEER
64-CRIBBAGE
SKIER-64
64 QUICK-CHART
SYNTHY-64

VIC TINY PILOT
VIC BUDGETEER
VIC VIGIL
VIC CRIBBAGE
GRAPHVICS

SCREEN DUMP
SPRITE-AID
VIC HIRES
VIC JOYSTICK PAINTER
VIC I-CHING

We are actively seeking SOFTWARE AUTHORS.
WHY NOT SEND US YOUR PROGRAM FOR EVALUATION.

Dealer Inquiries Invited
Write for our FREE Catalogue
for VIC and C-64

KING MICROWARE Ltd.

Suite 210,
5950 Côte des Neiges
Montreal, Quebec H3S 1Z6

Canadian manufacturer and distributor for ABACUS Software Products

Volume 5
Issue 02

Circulation 40,000

The Transition To Machine Language Editorial . 5

News BRK . . . 6

The Reference Transactor Is Coming!
Best Of The Transactor Volume 3 SOLD OUT!
Back Issue Quantity Orders
Subscription Problems
Department 'TR'
Commodore U.S. Updating Policy
Commodore International Announces New Microcomputers And Related Peripheral Devices
Commodore Receives Royal Warrant
Computer Song Writing Contest
Holt, Rinehart and Winston Now Publishing Software
SuperPET User's Group and the SuperPET Gazette
Tutorial Diskette For The SuperPET
Programming The PET/CBM
The Machine Language Book For The Commodore 64
The Anatomy Of The Commodore 64
The Anatomy Of The 1541 Disk Drive
MASTER-64
JOSEF – A New Programming Language
Application Software From Fabtronics
Music Production Service
Flexidraw's New 3.0 Version Opens Channel Of Communication
Flexidraw 3.0 Offers A Rainbow Of Colours
New Diskovery Early Math Programs
Turtle Toyland Jr. Teaches Basic Computer Concepts
'Horses OTB': Horse Race Handicapping Software
Commodore 64 Memory Expander
'Bit Scrubber' Disk Residual Noise Eraser
SADI Communications Interface and Printer Adapter
Electronic Fingerprint Analysis Security System

Letters 16

Un-products? (C64 Keyboard & Drums Synthesizer)
Response? Response: (Auto Liner Revisited)
Existing Lost Copy: (more on Program Generators)

Bits and Pieces 18

Kernal 3 For The Commodore 64
Cylinder Screen
Down Scroll 64
FTOUTSM With Colour Mods
Machine Language FTOUTSM
amaZAMARAing
Stop RUN/STOP
Cursed Commodore Cursor!
Sorry, But That DOES Compute
Low-Res Screen Copy
Eep Eep
Mirror
Ram Scan
Crystal
Number Base Converter
The Un-Cursor

CompuKinks 24
The MANAGER Column 27
Review: MailPro 64 34
Perspective: To GET Or Not To GET 36
All About Commodore Abbreviations 37
Messing With The Stack 49
The Un-Token Twins 51
Merging BASIC Programs 53
An Introduction To The Tools And Techniques Of Machine Language 55
Your BASIC Monitor 58
Finding Pi Experimentally 63
Translating A BASIC Program To Machine Language 66
A Few Of The Stranger 6502 Op Codes Explained 72
Getting BASIC To Communicate With Your Machine Code 76
CIA Timers 83
6526 Time Of Day Clock 86
Joycursor 90
Butterfield: SX64 Emulator 91

Advertising Section 92
Advertising Index 104

The Transactor
The Tech/News Journal For Commodore Computers

Managing Editor
Karl J. H. Hildon

Editor
Richard Evers

Advertising Manager
Kelly M. George
416 826 1662

Art Director
John Mostacci

Subscriptions
Mandy Sedgwick

Contributing Writers
Eric Armson
Don Bell
Michael Bertrand
Daniel Bingamon
Brad Bjorndahl
Jim Butterfield
Elizabeth Deal
Domenic Defrancisco
Bob Drake
Mike Forani
Jeff Goebel
Melissa Gibbins
Dave Gzik
Phil Honsinger
Mike Panning
Howy Parkins
Glen Pearce
Louis F. Sander
George Shirinian
Darren J. Spruyt
Colin Thompson
Mike Todd
Vikash Verma
James Whitewood
Chris Zamara

Production
Attic Typesetting Ltd.

Printing
Printed in Canada by
MacLean Hunter Printing

Program Listings In The Transactor

All programs listed in The Transactor will appear as they would on your screen in Upper/Lower case mode. To clarify two potential character mix-ups, zeroes will appear as '0' and the letter "o" will of course be in lower case. Secondly, the lower case L ('l') has a flat top as opposed to the number 1 which has an angled top.

Many programs will contain reverse video characters that represent cursor movements, colours, or function keys. These will also be shown exactly as they would appear on your screen, but they're listed here for reference. Also remember: CTRL-q within quotes is identical to a Cursor Down, et al.

Occasionally programs will contain lines that show consecutive spaces. Often the number of spaces you insert will not be critical to correct operation of the program. When it is, the required number of spaces will be shown. For example:

print" flush right" – would be shown as – print"[space10]flush right"

Cursor Characters For PET / CBM / VIC / 64

Down	– [q]	Insert	– [.]
Up	– [ ]	Delete	– [ ]
Right	– [ ]	Clear Scrn	– [ ]
Left	– [Lft]	Home	– [s]
RVS	– [r]	STOP	– [c]
RVS Off	– [ ]		

Colour Characters For VIC / 64

Black	– [P]	Orange	– [A]
White	– [e]	Brown	– [ ]
Red	– [ ]	Lt. Red	– [V]
Cyan	– [Cyn]	Grey 1	– [ ]
Purple	– [Pur]	Grey 2	– [X]
Green	– [ ]	Lt. Green	– [Y]
Blue	– [ ]	Lt. Blue	– [ ]
Yellow	– [Yel]	Grey 3	– [Gr3]

Function Keys For VIC / 64

F1 – [F]	F5 – [G]
F2 – [ ]	F6 – [K]
F3 – [ ]	F7 – [H]
F4 – [J]	F8 – [L]

The Transactor is published bi-monthly by Transactor Publishing Inc. It is in no way connected with Commodore Business Machines Ltd. or Commodore Incorporated. Commodore and Commodore product names (PET, CBM, VIC, 64) are registered trademarks of Commodore Inc.

Volume 5 Subscriptions: Canada $15 Cdn
U.S.A. $15 US.
All other $18 US.

Second Class Mail
Permit Pending

Send all subscriptions to: The Transactor, Subscriptions Department, 500 Steeles Avenue, Milton, Ontario, Canada, L9T 3P7, 416 876 4741. From Toronto call 826 1662. Note: Subscriptions are handled at this address ONLY.

Back Issues: $4.50 each. **SOLD OUT:** The Best of The Transactor Volume 3, Volume 4, Issues 4, 5, & 6 no longer available.

Quantity Orders:

ACCESS
Access Computer Services
630B Magnetic Drive
Downsview, Ontario, M3J 2C4
(416) 736 4402
Dealer Inquiries ONLY:
1 800 268 1238
Subscription related inquiries
are handled ONLY at Milton HQ

CompuLit
PO Box 352
Port Coquitlam, BC
V5C 4K6
604 464 1221

U.S.A. Distributor: Prairie News, 2404 West Hirsch,
Chicago, IL, 60622, (312) 384 5350

Want to advertise a product or service? Call or write for more information.

Editorial contributions are always welcome and will appear in the issue immediately following receipt. Remuneration is $40 per printed page. Preferred media is 2031, 4040, 8050, or 8250 diskettes with WordPro, WordCraft, Superscript, or SEQ text files. Program listings over 25 lines should be provided on disk or tape. Manuscripts should be typewritten, double spaced, with special characters or formats clearly marked. Photos of authors or equipment, and illustrations will be included with articles depending on quality. Diskettes, tapes and/or photos will be returned on request.

All material accepted becomes the property of The Transactor. All material is copyright by Transactor Publications Inc. Reproduction in any form without permission is in violation of applicable laws. Please re-confirm any permissions granted prior to this notice. Solicited material is accepted on an all rights basis only. Write to the subscriptions address above for a writers package.

The opinions expressed in contributed articles are not necessarily those of The Transactor. Although accuracy is a major objective, The Transactor cannot assume liability for errors in articles or programs.

From The Editor's Desk

The Transition To Machine Language

Why are so many afraid of Machine Language? Even the name is intimidating, like it's the dialect spoken on some silicon based planet in another galaxy or something. Perhaps we should give it some other name. Somehow I feel this wouldn't be enough, though.

So what is it? Is it the concept of reaching inside that thing called the microprocessor? Agreed, programming in raw hexadecimal can be a painful experience, one which nobody should be subjected to and still be expected to maintain enthusiasm. No, poking hex codes might be ok for the first couple of 5 byte programs, but any more and you'll soon be turned off.

Using a good Assembler will take the sting away. Most offer 6 character labels on variables and subroutines. Calling a subroutine by its name is much more practical than remembering *its number*. Still, most say even assembly language is too unsophisticated. . . no error messages, no string handling, no floating point variables, no this, no that. . . you have to do everything yourself!

Ok, I admit, there are things you can do in BASIC that would be hideously mind bending in machine language. Multiplication and division of fractions is one task I would cringe over, same with triginometry, and worse, calculus! But nobody said you should use machine language all the time. In fact BASIC is perfect when you need only a few calculations, a bit of file handling, and most printers can't go faster than BASIC anyways.

Except there are things you just can't accomplish within reasonable time in BASIC. Imagine a ten field, cascade sort. . . with enough data you wouldn't see the results in your own lifetime! This is where machine language truly makes its mark. Moving text around is a natural for machine code. And you don't need to 're-invent the wheel'. There are lots of sort routines and other machine language utilities around that you can usually just slip into your BASIC, and the crafty soon learn how to use ROM subroutines for some particularly nasty jobs. Even writing them yourself can often save you time in the long run. I don't need to remind you, machine language is lightning fast!

With the right tools, mixing machine language with BASIC is not only easy, but the results are much more rewarding. Yet we still avoid it. I believe one problem is the process which humans seem to naturally enjoy learning, that is, the easy way first. And too often *the easy way* is determined for us.

When I first learned to ski, I was taught how to 'snow plow'. Then when I tried parallel skiing, I found myself constantly reverting to snow plowing. It took a long time to break that habit. I think the same is true between BASIC and assembler. "Learn BASIC first, it'll give you a *feel* for programming", we're told. Then we get too comfortable in BASICs' care-free environment.

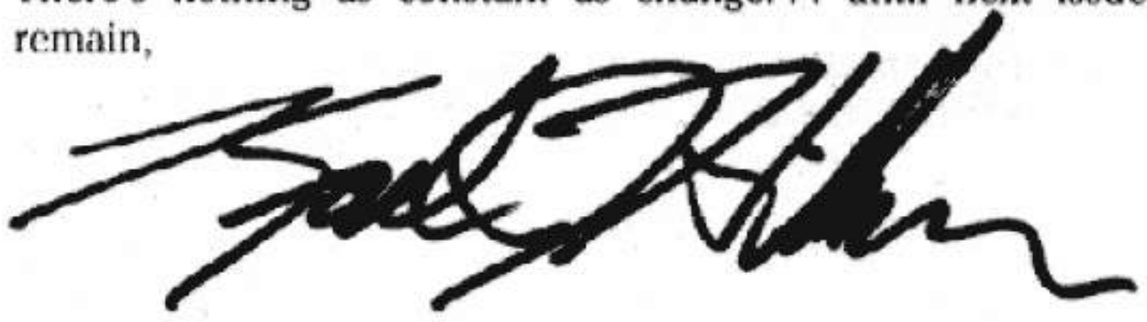
In my opinion, a students' first taste of computer programming should be a generous helping of machine language. With no prior experience of the high level approach, they would have nothing to compare against, and the apprehension would be eliminated. Only the fear of the unknown must then be overcome, which is true for learning anything new. After assembling a month or two of simple machine language efforts, unveil the high level interpreter and suddenly they gain new appreciation for programming. "RUN, you mean all I have to do is say RUN?" But with the order reversed, several new fears develop. Suddenly it's no longer possible to just say 'RUN', and the learner retreats.

Exposure to assembling machine language instills other disciplines too; pre-planning, variable definitions, correct structuring, and clear commenting and documentation, to name a few, are all necessary ingredients for machine language. They're important in BASIC too, but how many of you have deemed them 'unnecessary' at one time or another. Then, 6 months later, you need to add a variable to that program but don't know if it's already been used. Or you can't remember what that silly looking subroutine does at line 53427! High level environments lend themselves nicely to side-stepping. Those that acquire a little machine language discipline early will naturally apply it when the time comes to write in BASIC. And when that machine language subroutine just can't be substituted, it will be treated like a natural step towards completion, rather than a job to be procrastinated as long as possible.

I urge you to try your hand at machine language or assembler. Read through the listings in this issue or disassemble some code in a program you're using. Supermon has a disassembler in it and it's available from TPUG and most club libraries.

Your computer is a creature of machine language. BASIC is there merely to tame the creature. Well it's time to cast away the protective shield and confront the creature on its own turf. You'll find the challenge of the microprocessor is is no greater than any other challenge, and once you conquer it, you'll be classifying BASIC as an unworthy opponent!

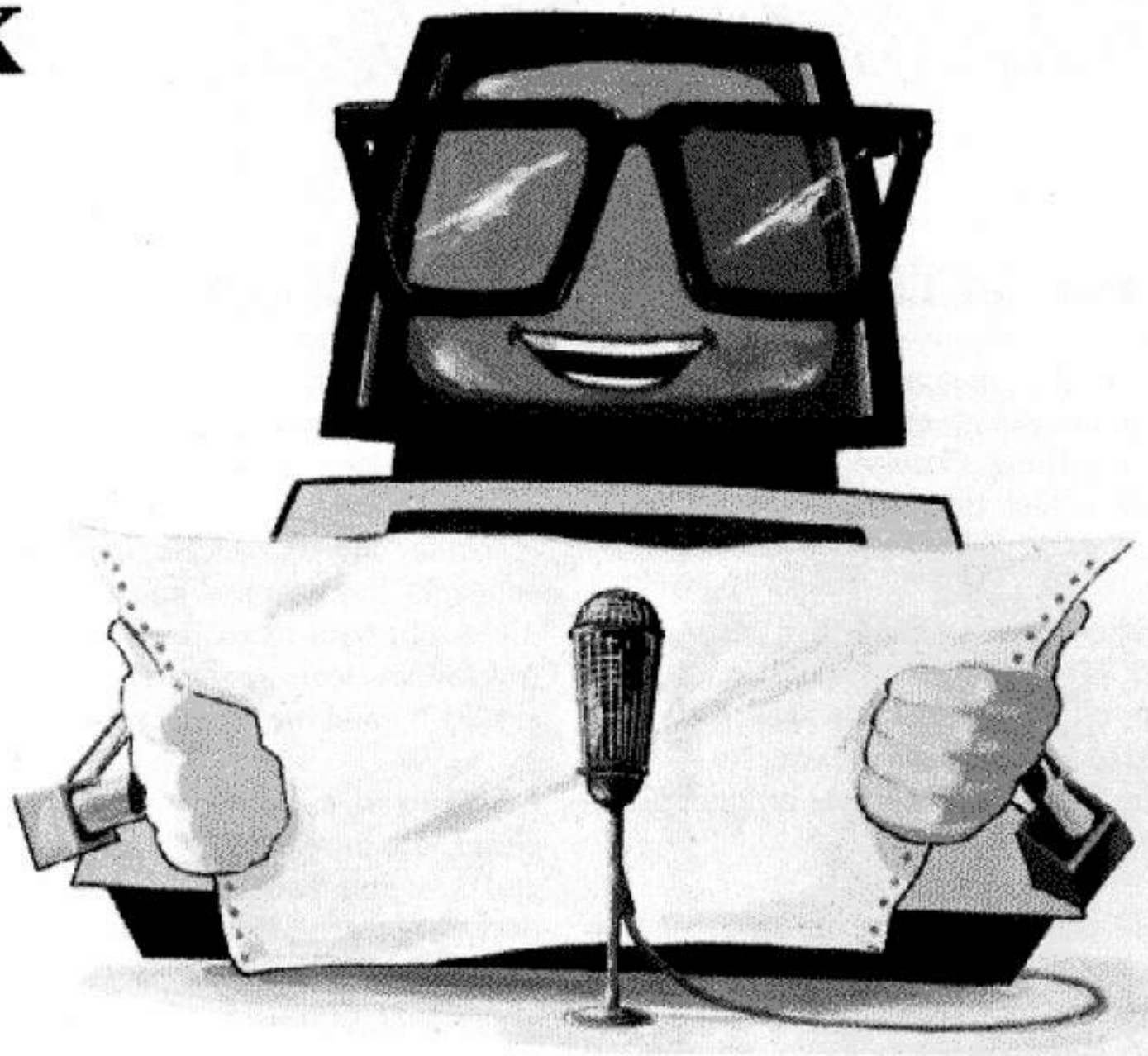
There's nothing as constant as change. . . until next issue, I remain,



Karl J.H. Hildon
Managing Editor, The Transactor

Post Script: Notice our circulation figures? Almost double our last number! Consequently we couldn't get enough Concorde book paper in time for this issue so it had to be printed on this shiny stuff. We were disappointed too, but we'll be back printing on the quality stuff next issue!

News BRK



Transactor News

The Reference Transactor Is Coming!

We've been getting so much response over The Reference Issue (Volume 4, Issue 05) that we've simply run out trying to meet the demand. Reference Issue "Users" have reported buying every copy in the store just to have extras when one wears out. Others have spent several dollars getting every page laminated. On behalf of The T., I'd like to thank everyone for their compliments regarding Issue 05 and particularly those who have forwarded corrections and suggestions for improvements.

Although we will not be re-printing Volume 4, Issue 05, the NEW Reference Transactor is already in motion. Scheduled to be released around September 1st, 1984, it will be offered as a separate item from Transactor Publishing Inc. and will therefore not be included with a regular subscription. The book will be printed on fine quality #1 book paper and "ring bound" so it will lie flat when opened to any page. Cover pages will be made of a thicker stock to prevent untimely wear. Several new sections are planned including a complete list of PEEK and POKE procedures for all of memory, an expanded Glossary and book list, disk memory maps, plus all memory maps and pertinent data for the new Commodore equipment that should be on the market by then.

The price will be somewhere between $12.95 and $24.95. A rather large spread I agree. The reason? We may include a diskette sewn into the cover of every book. The disk would contain copies of any utility type subroutines listed inside, plus a whole batch of programs that get a lot of use around The Transactor development department. Programs like Supermon (versions for all models), assemblers and un-assemblers, disk monitors, file readers, data generators, intelligent directory readers, games utilities, sort routines, copy programs, programmers aids, communications software, text editors, plus anything else we can dig up that comes in handy when you need it most. And with some 7 years of collecting these programs, we'll have plenty to dig through!

Please do not submit orders yet. This time we plan to print enough so there will be plenty to go around. Final details, including quantity orders, will be released in our next issue (August '84).

Best Of The Transactor Volume 3 SOLD OUT!

Our current subscription cards show a space for ordering "The Best Of The Transactor Volume 3". Please do not complete this section as they too are all sold out. A "Best Of The Transactor Volumes 1 to 4" has been considered, but no firm plans have been set for its production so, once again, please do not submit orders yet. It may also sport a diskette of programs contained within. . . details to follow next issue.

Back Issue Quantity Orders

Please note that back issues are to be ordered from The Transactor head office in Milton, Ontario, and the current issue from our distributor nearest you. The current issue becomes a back issue as soon as the next issue is released. At this point our distributors return the unused portion of their shipment. Therefore you can only get them from us.

Recently we've been receiving several orders for quantities of back issues from US retailers. However, between shipping charges, the quantity discount, and customs service charges, we actually lose on the transaction. In light of this situation, quantity orders from the USA for back issues will be costed at 25% of retail value plus $40.00 for postage, handling, and customs surcharges.

Subscription Problems

If you have a problem with receiving magazines, we want it fixed as quickly as you do! To help speed the process, please write us with a complete explanation of the situation. If you paid by cheque, send us a copy - not that we don't believe you've paid, it just helps us find you easier and track the problem to the source.

If you have renewed your subscription and find you are receiving two of each issue, then you've probably been entered twice in our mailing list system. In this case your subscription has been duplicated when it should have been extended. This is our mistake, but if not reported, you will receive less issues than you are entitled to. If you're currently receiving duplicate issues that you haven't ordered, please inform us so we can correct it. And please keep the duplicates with our apologies.

Department 'TR'

Each address given in the News BRK section will now include the line, "Dept. TR." Please include this in the address should you write for more information on a product listed here. It saves you the trouble of noting where you found the initial product information (which we appreciate nonetheless) and it also gives the manufacturer (or maybe potential advertiser?) an idea of how much response The Transactor might generate for them (which we also appreciate).

Commodore News

Commodore U.S. Updating Policy

To receive an update for any of Commodore's software, send $5.00 U.S. plus the original diskette, along with the replacement diskette card or a receipt to;

Commodore Disk Replacement
1200 Wilson Drive
West Chester, PA 19380

Commodore International Announces New Microcomputers And Related Peripheral Devices

NEW YORK, N.Y. — Commodore International Limited (NYSE:CBU) introduced a new line of microcomputers, related peripheral devices and accessories, many of which have never been shown publicly. The new introductions were made at the Hannover Fair in Hannover, West Germany.

Among the new products publicly shown for the very first time anywhere in the world were two new microcomputer systems for the business market.

The first system is a 16–bit, Z8000 microprocessor based computer system which features a Unix–oriented operating system, 256K bytes, or 256,000 characters of built–in user RAM, or random access memory, 80 column colour graphics, and built–in dual floppy disk drives. Optional hard disks and printers will also be available for this new system.

The second system, a 16 bit, 8088 microprocessor based Commodore PC, is a transportable system with software compatibility with the IBM Personal Computer, and also includes 256K bytes of built–in user RAM. This system has more features and will be sold at a lower price than the IBM Personal Computer.

In addition to the two new business systems, other computers shown at Hannover were two microcomputers designed for the home market, the Commodore 16, featuring 16K bytes, or 16,000 characters of built–in user RAM, and the Commodore 264, the series name for a 64K microcomputer that ws originally introduced at the Consumer Electronics Show in Las Vegas, Nevada in early January.

Among printers and other accessories introduced at the Hannover Fair were four new printers designed for the VIC–20, the Commodore 64, and the new 264 series. These include a low cost dot matrix printer, a higher–end dot matrix printer, a colour dot matrix printer, and a low cost daisy wheel printer.

Finally, Commodore also introduced three new accessories for the 64, including a touch screen, a light pen and the Commodore CAT, a mouse–like device.

All products, except the two business systems, will be available during the last half of 1984, while delivery dates for the business systems will be announced. Contact:

Mr. Steven A. Greenberg
30 Rockefeller Plaza
NEW YORK, N.Y. 10112.
(212) 246–1000

Commodore Receives Royal Warrant

Commodore Business Machines (UK) Limited, the leading microcomputer manufacturer, has become the first manufacturing company to be granted the Royal Warrant of Appointment by Her Majesty The Queen of England for computer business systems.

General Manager of Commodore UK, Mr. Howard Stanworth, said: "As a high technology company with a growing manufacturing and ancillary supplier base in the UK we are delighted and honoured to receive the Royal Warrant of Appointment to Her Majesty The Queen."

The Warrant carries the legend "By Appointment to Her Majesty The Queen, manufacturers of Computer Business Systems, Commodore Business Machines (UK) Limited Slough" and is for an initial period of ten years.

Commodore UK is based at Ajax Avenue, Slough, England and has a factory at Corby, Northants, which produces more than 5,000 microcomputers a day.

The Company currently employs a total of more than 300 people and later this year will create up to 1000 jobs when it opens its European manufacturing and distribution headquarters at Corby.

Its best known products include the Commodore 8000 series systems, the VIC 20 home computer and the Commodore 64, recently voted "Home Computer of the Year" by a number of international computer journals. For further information, please contact:

Mr. Wu Yhee Ming (Managing Director)
Systems Technology Pte. Ltd.
149 Rochor Road #04–10
Fu Lu Shou Complex
Singapore 0718

General News

Computer Song Writing Contest

Vince Fleming of Strangeland Music (ASCAP) and Dan Seitz of Aleph–Baze Music (BMI) have announced they will join the panel of judges for EnTech Software's First Annual Computer Song Writing Contest. Aleph–Baze and Strangeland are music publishing companies in the Los Angeles area. Other contest judges will be named soon, and may possibly include executives from CBS and Capitol Records.

EnTech's Computer Song Writing Contest, the first of its kind, will award $1,000 and free studio time to the best musical composition written on the Commodore 64 with EnTech's "Studio 64". Studio musicians, an arranger, and a producer will help turn the winner's composition into a hit song.

Contest entry blanks are available at participating dealers, and entries will be accepted through November 1, 1984. For more information, contact:

Mathew Stern
ENTECH Computer Song Writing Contest
PO Box 185
Dept 'TR'
Sun Valley, CA 91353
818 768–6646.

Holt, Rinehart and Winston Now Publishing Software

Canadian book publisher Holt, Rinehart & Winston has made the move to publishing software; one of the first of the large publishing firms to do so in this country. Already a distributor of CBS software, as well as Compute!, Hayden Books and dilithium Press computer books and software, HRW is going to develop, manufacture and sell software here in Canada.

"What we intend to offer", said Carl Cross, Vice President of the Trade and Professional Division, "are quality products for the home educational and business applications market. We don't want any 'three month wonders'; we're looking at solid items with a longer selling life and strong backlist potential.

Carl also said that, while HRW's emphasis would be on Canadian authors and content, they would also be looking for software which has potential for international marketing, particularly in the United States where their connections with CBS guarantee them a large market reach.

HRW intends to bring out its first product, already under development inhouse, in early summer of this year. Others will follow throughout the year. They are currently assembling a staff to manage the projects and have hired Ian Chadwick, former editor of InfoAge Magazine, freelance writer and author of "Mapping the Atari", as Software Editor.

Rather than hiring a large staff base of programmers and developers, Holt will work with a wide variety of talented, outside resource people. They are interested in discussing projects with any Canadian programmers, authors, teachers or developers who feel they can contribute to the developement of software for the popular home and business microcomputers, including the Apple IIe, MacIntosh, IBM PC and PCjr, Commodore 64, Atari 800XL and others. Interested parties should contact Carl Cross or Ian Chadwick at (416) 255–4491 during normal business hours.

Holt, Rinehart and Winston of Canada, Limited
Dept 'TR'
55 Horner Avenue
Toronto, Ontario M8Z 4X6

SuperPET News

SuperPET User's Group and the SuperPET Gazette

The SuperPET User's Group, with members from Canada, the U.S., and Europe, has commenced publication of "The SuperPET Gazette". This newsletter aims to provide valuable information on using the SuperPET. For example, the September 1982 issue included articles on the following subjects:

Waterloo microBASIC: The Keyboard and its Codes
Using BASIC Procedures in Immediate Mode
SuperPET News

A free copy of the September issue, which contains information on how to become a SuperPET User's Group member, can be obtained by sending a request along with a U.S. 20–cent stamp to:

The Editor
SuperPET Gazette
PO Box 411
Dept 'TR'
Hatteras, NC 27943

Tutorial Diskette For The SuperPET

This product contains five tutorial files divided into 19 sections, describing a variety of aspects of the SuperPET capabilities. The most central facility of the SuperPET is the microEDITOR, but unfortunately it is also the facility which is least well documented. This tutorial disk discusses all uses and commands of the editor and could be considered an equivalent to the programming–language examples supplied by Waterloo on the tutorial disk. This tutorial also contains much reference material not provided by Waterloo or Commodore. Many of the facilities described will only work under version 1.1 (and hopefully any later versions) of the Waterloo software.

This diskette contains this DESCRIPTION file, a CONTENTS file which is a table of contents for the tutorial sections; TUTORIAL files numbered 1 through 5 containing the actual tutorial; and a LICENSE file which explains the product warranty and conditions of use. In addition there are some public–domain programs distributed as a courtesy. All the contents of the disk are briefly described in the DIRECTORY file.

The topics in the tutorial files are not presented in any particular order, except to some extent from more general to more 'technical'. It would be possible to read the entire 5 tutorials in order, (although with 10000+ words it would take some time! but a better approach might be to start with the topics of known interest

and delve into the others as the need arises. You might want to extract out into other files or print out some subsets of the tutorials for quick reference. If you have the Waterloo !Help! facility (and a dual drive) you could use it to automate scrolling through the tutorial files.

The CONTENTS file shows the file name and exact line number of each section header. You can go to a particular section by just typing this line number after GETting the file, or of course you can always find the next section header with a +/SEC command. Of course you should try out the various edit facilities while you are going through the tutorial – you will learn to use the SuperPET facilities by USING them. Naturally you should start by creating a backup copy of the tutorial disk before you do anything else (copy programs are supplied for that purpose).

We wish you the best of luck in mastering your Commodore SuperPET. It is one of the most sophisticated micro systems available and an excellent vehicle for learning to compute. Do try to do what you can on your own, but you should feel free to call on help. The BIBLIOGRAPHY file will lead you to some groups and other sources as well as books. Feel free to write to this address if you have questions:

Dyadic Resources Corporation
PO Box 1524, Stn. 'A',
Dept 'TR'
Vancouver, BC V6C 2P7

Books

Programming The PET/CBM

The UK edition of 'Programming the PET/CBM' is available in the US (after Compute! discontinued printing) from this address:

Holford Enterprises
6065 Roswell Road
Suite 1398
Dept 'TR'
Atlanta, GA 30328

The Machine Language Book For The Commodore 64

The Machine Language Book For The Commodore 64 is aimed at the Commodore 64 owner who wants to progress beyond BASIC. If the reader wants to write programs that run faster, use less memory or perform functions that are not available in BASIC, then this book will help him understand machine language.

This is a 200+ page detailed guide to the complete instruction set of the 6510 processor of the Commodore 64. The book is filled with examples of machine language routines so that the reader can learn from working programs. These examples are geared specifically to architecture of the Commodore 64.

Included in these pages are listings of three full length programs. One is a working assembler so the reader can create his own machine language programs. The second is a working disassembler so the reader can inspect other machine language programs. The third is a 6510 simulator so that the reader can better understand the operation of the processor.

The Machine Language Book For The Commodore 64 is scheduled for release in April in softcover for $19.95. Available from your local dealer or directly from Abacus Software.

Abacus Software
PO Box 7211
Dept 'TR'
Grand Rapids, MI 49510
616 241-5510

The Anatomy Of The Commodore 64

The Anatomy Of The Commodore 64 is aimed at the Commodore 64 owner who wants to better understand his micro. It is a 300 page detailed guide to the lesser known features of the 64. Here's an outline of the contents:

1. Machine Language Programming On The Commodore 64
2. The Next Step – Assembler Language Programming
3. A Close-Up Look At The Commodore 64
4. Music Synthesizer Programming
5. Graphics Programming
6. BASIC From A Different Viewpoint
7. Comparison Of The VIC-20 And The Commodore 64
8. Input And Output Control
9. ROM Listings

Those readers that need to delve deeply into their computer, we've included a fully commented listing of the ROMS. Here's an authoritative source for Commodore 64 information.

The Anatomy Of The Commodore 64 in softcover $19.95. Available from your local dealer or directly from Abacus Software.

The Anatomy Of The 1541 Disk Drive

The Anatomy Of The 1541 Disk Drive is aimed at the Commodore 64 owner who wants to better understand his disk drive. It is a 300+ page detailed guide that explains the mysteries of using the floppy disk. Here's an outline of the contents:

1. Getting Started
2. Storing Programs On Disk
3. Disk Commands
4. Sequential Data Storage
5. Relative Data Storage
6. Disk Error Messages
7. Direct Access Commands
8. Overview Of DOS Operation
9. Structure Of A Diskette
10. Utiltiy Programs
11. 1541 ROM Listings

If you've been confused about using files on the 1541 then this guide clearly explains their use with many examples. We've also included listings of many useful utilities that you can use including a DISK MONITOR.

Those readers that need to delve deeply into their disk drive, we've included a fully commented listing of the 1541 ROMS. Here's the authoritative source for 1541 Disk Drive Information.

The Anatomy Of The 1541 Disk Drive scheduled for March 1984 in softcover $19.95. Available from your local dealer or directly from Abacus Software.

Software News

MASTER-64

MASTER-64 is simply the best, most comprehensive professional application program development package. No other software package offers near the features of MASTER-64. MASTER-64 has commands for programmer's aid, screen management, superior indexed file management, high multiprecision arithmetic, machine language monitor and much more. And software that you develop using MASTER-64 can be distributed without paying royalties.

MASTER-64 adds almost 100 new commands to BASIC that include:

SCREEN MANAGEMENT - define, input, edit and output data in exacting format to/from screen. Save, load or swap predefined screen.
ISAM FILE SYSTEM - complete support of up to 10 indexed sequential files. Data packing gives up to 40% more data storage. Fast indexed or sequential retrieval.
PRINTER GENERATION - define and format printer pages similar to screen management.
BASIC EXTENSIONS - multi-precision (22 digits) arithmetic, direct disk access, date control, more.
PROGRAMMER'S AID - auto, renumber, delete, print using, find, if then else, trace, dump, error, etc.
BASIC 4.0 COMMANDS - for compatibility with other Commodore micros. Includes relative record access.
MACHINE LANGUAGE MONITOR - built into MASTER-64 for added usefulness.

For serious programming development, nothing comes close to the power of MASTER-64. MASTER-64 comes complete with a comprehensive 160 page user's manual in three-ring binder, the MASTER-64 development system and the MASTER-64 runtime package.

MASTER-64 Software & Manual on diskette $84.95. Available now from your local dealer or directly from Abacus Software

JOSEF - A New Programming Language

JOSEF is a new powerful educational programming language for microcomputers which combines the spirit of the Logo Turtle with the structure of Pascal. With the Turtle it shares extendability (programs become new language words), the challenging yet toy-like and natural programming environment with visual orientation, and the possibility to execute commands directly without the need to write programs.

JOSEF has standard programming features such as variable, assignment and i/o statements, control structures, procedures and functions with parameters and recursion, as well as unusual constructs such as programmable interrupts allowing the writing of games controlled from the keyboard. A consequence of the screen oriented nature of the language is that pseudo-graphics is possible without special hardware.

The latest Version 1.1 contains a built-in interactive tutorial that allows the user to learn the language directly from and in interaction with the computer.

JOSEF is a robot who uses the screen of an ordinary terminal or microcomputer as a geographical map of his world and can be programmed to perform natural everyday tasks. He can move on the map, write on it, manipulate user-created objects, communicate, sense information about the map, and so on. The program contains a map editor which allows the user to create his/her own maps.

JOSEF is intended for people who want to learn about programming in an interesting and natural way, children, but also for mature programmers looking for something different. JOSEF is ideal for education, particularly because of the increasing emphasis on teaching Pascal in high school programs.

The program runs on a number of computers with enough memory and sufficient disk drive capacity. The cost is $45 for individual users and $65 for schools (permission to make multiple copies).

A textbook for the language called The First Book of JOSEF (list price $13.95) by Ivan Tomek and published by Prentice-Hall is available in bookstores and from Kobetek Systems.

Designed by:	Distributed by:
Modular Systems 82 POB 1456, Wolfville, Nova Scotia, Canada B0P 1X0	Kobetek Systems 1113 Commercial Str., New Minas Dept 'TR' Nova Scotia, B4N 3E6 902 678-7771

Application Software From Fabtronics

Utiltiy File - VIC 20 (+3K)/C64 - Tape/Disk $27.95 (formerly "Energy Master"). Extensive energy consumption data processing program to calculate, display, store, print out data including daily averages/totals and cost projections for any number of days.

Electric: K.W.H.
Water: Gals/cu-ft (automatic conversion)
Gas: cu-ft/meters
Oil: Gals/litres
Propane: Pounds

Special Features - prior meter readings are retrieved within automatically, also any utility not applicable is bypassed.

Utility File II - C64 Disk $87.95. A commercial version of above with accounts payable including statements and billing on selected commercial forms.

Fill-A-Form - VIC 20/C64 - Tape $17.95 Disk $19.95. A numerous selection of in house plain paper business/commercial forms to help the private enterpreneur. Printer required.

Tenant File – VIC 20/C64 – Tape/Disk $19.95. Maintain a record on each tenant with 22 fields of information including: active or closed, name, rental rate, rent due day, unit # tenant in, social security #, residence phone #, business phone #, auto yr/make, plate #, driver lic #, sec. dep amt paid, sec. dep date paid, sec. dep amt returned, retention reason, date moved in, date moved out, # of bad checks, condition of unit on move in, condition of unit on move out, comments/information pertinent to tenant or file

Rental Manager – C64 – Disk $47.95. Spreadsheet & data records for rental applications dte oriented tracking/billing and recording of payments. Also prints statements will support the following entries: account #, rate, unit or item #, payment schedule, due date, deposits, payments (10 per mo), utility charges, misc. chrgs, dates in, dates out, name of occupant or user

Fab Mail – VIC 20/C64 – Tape $16.95 Disk $19.95. A super user friendly mailing list with features others wish they had thought of, ez select/edit, user selection of gemini 10x and similar printer abilities, also great for cataloging/filing/sorting.

Fab Business – C64 – Disk $47.95. A mail order or small business must. Easy invoice/packing list/label all in one, supports charge card data and allows quick selection of items when used with "inventory d–base"program. "Fab–Mail" data also quickly retrieved. Plain paper or selected comm. forms.

Inventory D–Base – C64 – Disk $27.95. A stand alone program that is also compatible with fab–business. Allows quick selection of items during invoicing.

Printing on all programs where applicable are compatible with commodore 1525, MPS801, Epson, Gemini, Prowriter, C–Itoh, Epson MX/RX/FX 80–100, Okidata 82/83/84/92/94, Axiom GP/100, Gorilla Banana and similar printers when properly interfaced.

S&H $3.00 — Visa/MC welcome no surcharge — all prices U.S. funds – (cash)o.d. to U.S. only add $2.00. NY add sales tax

Music Production Service

I am introducing a professional music production service for educational and game programmers at all levels who are interested in putting together the most attractive possible software for the Commodore 64 in order to stand out in todays's competitive market.

Music is a powerful tool for communication. Think of the majestic main theme from the "Star Wars" epics, or of Woody Woodpecker's laugh at the beginning of a cartoon. Obviously music can provide an emotional response as immediate and as strong as a visual display alone, and when the two are combined, the effect which results can be potent.

The sound capability of the Commodore 64 is extensive. However, without expertise in digital sound synthesis, three–voice harmony and counterpoint, specialized systems of intonation, musical copyright law, composition, and arranging, it can be difficult to produce high quality music.

With 20 years of experience as a composer/arranger, performing musician, and music educator, I can help with virtually any aspect of musical programming for the Commodore 64, up to and including complete interactive soundtracks for software. I have written for big bands, rock bands, school bands, club bands, brass quintets, woodwind ensembles, soloists, and now computers. I have played principal trumpet with the St. Pul Chamber Orchestra, the San Jose Symphony, and the Marin Symphony, played on records with Dave Brubeck and the St. Paul Chamber Orchestra, and played on many TV and radio shows and commercials, I have taught for the University of Minnesota and Macalester College, given clinics and master classes, and taught hundreds of private students.

If you would like an audio tape demonstrating a variety of different kinds of music that can be produced on the Commodore 64, send $4.00 to Tom Jeffries, 2915 Harrison, Oakland, CA 94611. Further information:

Tom Jeffries
2915 Harrison
Dept 'TR'
Oakland, CA 94611
415 451–3314

Flexidraw's New 3.0 Version Opens Channel Of Communication

San Diego, CA — Inkwell Systems increased the versatility of Flexidraw; graphics software and light pen, to include a communication program which enables two C–64 owners to send and receive graphics and text created by Flexidraw via modem.

This program, titled Transgraph, is easily accessed from a light pen driven menu and implemented by a series of on–screen user–friendly prompts. Any Flexidraw file can be sent or received using the VIC or HES modem, saved to disk and printed at both locations.

Sherry Kuzara, President of Inkwell Systems said "Transgraph represents a major step in the evolution of affordable business and personal communication." According to Kuzara, the network capabilities of Transgraph coupled with the "print out" feature can fill many business and personal needs, previously only available in systems costing thousands of dollars.

Flexidraw will still retail for $149.95. Registered owners will be notified by mail and can update their Flexidraw 2.1 version for a nominal fee. Distributor and dealer inquiries can be directed to Inkwell Systems:

Mr. Byrne Elliott/Ms. Sherry Kuzara
Inkwell Systems
P.O. Box 85152 MB290
7770 Vickers St., #202
Dept 'TR'
San Diego, CA 92138
619 268–8792

Flexidraw 3.0 Offers A Rainbow Of Colours

San Diego, CA — March 20, 1984, Inkwell Systems has expanded the capabilities of Flexidraw; graphics software and light pen combination, to include an interactive high resolution colour

program. Graphics previously created with Flexidraw can now be painted in a dazzling array of 16 high-resolution colours.

Completely light pen and menu driven, the new addition to Flexidraw version 3.0 entitled Pen Palette, features a similar screen/menu format as Flexidraw. With the ease of an artist using a paint brush and palette, the user chooses colour combinations from a series of paint pots located on the menu and applys these colours to areas on the work screen using the light pen.

Special features found on Pen Palette include; two demonstration programs illustrating colour animation capabilities, and the ability to save colour files to disk or re-load using the light pen driven menu selector.

Flexidraw will still retail for $149.95. Registered owners will be notified by mail and can update their Flexidraw 2.1 version for a nominal fee. Distributor and dealer inquiries can be directed to Inkwell Systems.

New Diskovery Early Math Programs

Internatinal Publishing & Software, Inc., announces the release of TAKE-AWAY ZOO and The ADDING MACHINE; two early math learning programs for 4 to 8 year olds. Take-Away Zoo and Adding Machine are the latest additions to I.P.S.'s successful DISKOVERY Learning Works line. Both use humorous graphics, sound and colour to create the fun and excitement needed to insure high student interest and long lasting learning.

These two new programs each consist of a set of three different but related activities designed to help the child become more successful in arithmetic at school. As the child helps the "animal-master" move animals in and out of their cages, he/she learns and practices addition and subtraction.

Each aspect of each activity is user controlled to allow the child to set the pace of the learning. These math activities are designed to give every child a successful addition or subtraction learning experience. While the activities are fun and exciting to play, they provide a constant challenge for the children.

ADDING MACHINE and TAKE-AWAY ZOO feature the DISKOVERY ELECTRIC REPORT CARD. This user-transparent special feature tells the teacher or parent (or child) the activities the child used, errors made (actually showing the problems answered incorrectly), final score and percentage right. The ELECTRONIC REPORT CARD is on, constantly providing complete up-to-date information on the child's progress at any time; even in the middle of an activity.

Each program teaches, tests, reviews and scores in an exciting game format which helps and encourages children to practice school subjects at home.

DISKOVERY Learning Materials are written and designed by leading educators ensuring that each skill taught matches the school curriculum.

TAKE-AWAY ZOO and THE ADDING MACHINE are currently available for the Commodore 64 and the Timex Sinclair 2068 computers. Apple II and IIE and TRS80 versions will be available by mid-June, 1984. The suggested Retail Price for disk versions of either program is $29.95. Dealer and distributor enquires are welcome. Write to:

International Publishing & Software Inc.
3948 Chesswood Drive
Dept 'TR'
Downsview, Ontario M3J 2W6
416 636-9409

Turtle Toyland Jr.
Teaches Basic Computer Concepts

For children aged six and up, the challenge of learning about computers and computer concepts has been made easier, and a lot more fun, with the introduction of Turtle Toyland Jr. by Human Engineered Software of Brisbane California, and distributed by Micron Distributing.

Available for the Commodore 64, IBM Personal Computer and Coleco systems, Turtle Toyland Jr. operates with just a joystick, teaching children computer concepts by moving a turtle across the computer screen to build film strips.

Turtle Toyland Jr. is an ideal introduction to creative programming for young children. Because the program translates a child's joystick movements into reproducible turtle graphics, children learn programming concepts and techniques.

To achieve the best results from the game it is recommended that a carefully designed sequence of activities is followed, beginning with a playground to discover how to move the turtle and draw images. From this introductory phase, children move on to turtle training and then on to the Crossroads to decide where to go next.

After a stop in Training Land, children can try four other sequences: Music Land, Sprite Land, the Roybox and Input/Output Land. In Music Land, children can learn to write their own music using the joystick to control notes from a piano, horn, guitar and flute. The music created can be stored in the Toybox.

Children in Sprite Land fill in squares with the turtle to draw sprites, which are animated drawings. Sprites can also be stored in the Toybox. In Input/Output Land, saved files in the Toybox can be called up and played again.

Turtle Toyland Jr. was developed jointly by Human Engineered Software and Childware Corporatin, an innovative software development group.

These and other innovative educational programs are distributed through Micron Distributing or can be found at your local computer store.

Micron Distributing
409 Queen St. West,
Dept 'TR'
Toronto, Ontario, M5V 2A5
416 593-9862

'Horses OTB': Horse Race Handicapping Software

Horses OTB is a thoroughbred horse race handicapping program. This program can be used for off track betting (OTB).

Statistics from a large number of races are combined with computer simulation methods to calculate optimum betting strategies. Data complied from over 1000 races shows superior performance from this system.

The program is easy to use. The computer asks questions about each horse in each race. The user needs to obtain the "Daily Racing Form" and answer the questions using the data from the form. The computer will tell the user which horses to bet on. No judgement or comparison of odds is necessary and the user does not need to know the track odds.

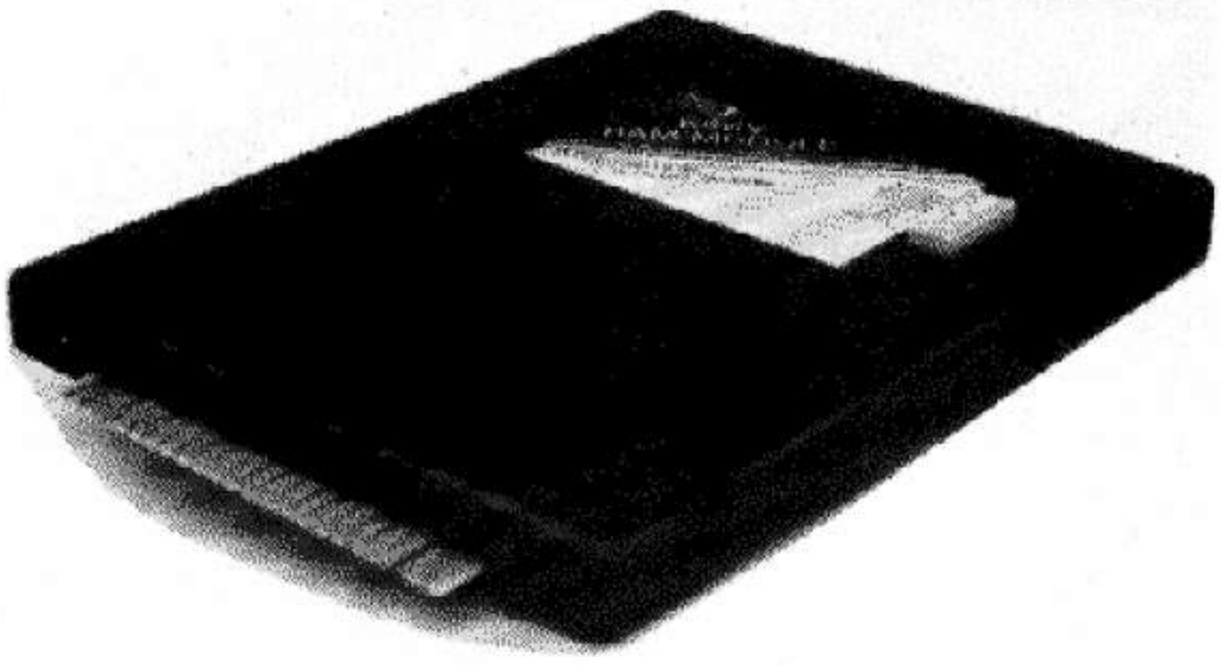
The complete instruction manual includes an explanation on how to use Horses OTB, facts on all input data needed and how to correct any mistakes on the data entered. The manual also contains tips on money management and presents simulated results of the money management techniques.

Horses OTB sells for $34.95 and is available on disk for the Commodore 64. It can be ordered by mail or through local dealers. Or contact:

Jim Golts
3G Company, Incorporated
RT3, Box 28A
Dept 'TR'
Gaston, Oregon 97119
503 662-4492

Commodore 64 Memory Expander

The unexpandable memory configuration of the Commodore 64 is no longer unexpandable! LETCO, the pioneer of the popular 64K memory expander for the VIC 20, announces the adapter (Model 64KVA) to use with their 64KV memory expander on the Commodore 64. When used on the C-64, the addresses from $8000 to $9FFF will have 8 separate blocks of 8K locations, each block selected by a single poke instruction. Current owners of the 64KV only need the adapter to use their memory on the C-64.



The adapter (Model 64KVA) is priced at $29.95. The memory (Model 64KV) for use on the VIC 20, is priced at $109.95. The combination (Model 64KVA) for use on the C-64, is priced at $139.95. Complete instructions are included with each product. All products are covered by a 90 day warranty on parts and labor and of course satisfaction is guaranteed within a 15 day return period.

LETCO will soon announce the ultimate expander for the 64, that will allow up to 256K bytes of expansion. Of course, current products will be compatible and pricing is expected to be about $140 per 64K byte module.

LETCO is currently working with many popular software suppliers to incorporate these added capabilities to their current and future releases. Just think of the power that can be added to your word processer or spreadsheet programs. All products are available directly from:

LETCO
Leader Electronic Technology Company
7310 Wells Road
Dept 'TR'
Plain City, OH 43064
614 873-4410

'Bit Scrubber': Disk Residual Noise Eraser

Now - Wake up that old pile of diskettes you don't want to throw away! Every computer has a common purpose—knowledge stored in program format. But when dealing with giga and mega kilobytes of data, one single bit erroneously entered can render a program useless, create frustration, time delay and profit loss. Editing a program also generates on-disk magnetic clutter causing a display of "error messages", "disk overload", etc.

In addition, power supply fluctuations and disks remaining in the computer during system power down, produce a magnetic field around the drive head correspondent to these currents, generating even more noise. When the head attempts to read/write data from this portion of the disk, it cannot, and eventually this affects the entire file.

During normal use, as a data file is revised, the head erases previously stored data and replaces it with new data. This erasing process is not perfect, always leaving a trace of magnetic noise. After many write/erase operations, this noise level justifies thorough disk erasure.

Disk storage media replacement is expensive and time consuming. For these reasons, Techstar, Inc. has developed the "Bit Scrubber". The fastest and most positive method to magnetically "clean" and standardize both new and used diskettes.

The "Bit Scrubber" can restore used and noisy disks to their original magnetic quality, providing a cost efficient method of "error-free" storage eliminating the expense of purchasing new floppy disks.

The "Bit Scrubber" can reclaim and maintain "SSSD" (single-sided, single density), "SSDD" (single-sided, double density), "DSDD" (double-sided, double density), "SSQD", "DSQD", etc. and any other type of commercially available floppy disk.

When used periodically, the "Bit Scrubber" prevents noise accu-

mulation on the disk, assuring reliable data storage and extending the life of this costly recording medium.

- "Bit Scrubber" will clean 8", 5 1/4" and the new mini diskettes.
- Patented high energy magnetic "gap" insures uniform particle orientation.
- Shielded magnetic circuit protects programmed disks.
- Dimensions: 9" x 4" x 1–3/4", Weight: 5 lbs.

$49.95 – plus $4.00 shipping and handling charges (Fla. residents add 5% sales tax)

Techstar Inc.
8651 N.W. 56th Street
Miami, FL 33166
Dept 'TR'
305 592–0201

SADI Communications Interface and Printer Adapter

The cMc SADI is a microprocessor based interface designed to allow communication between Commodore PET and CBM computers and a wide range of devices including serial and parallel printers, CRTs, modems, acoustic couplers, hardcopy terminals nd other computers. SADI's two independent ports (one serial in/out and one parallel out) give the Commodore computers tremendous flexibility as controllers and as dumb or smart terminals. Data can travel between the computer and one or both ports or between ports.

General features include true ASCII conversion, cursor move conversions for program listings, and automatic insertion or deletion of linefeeds. The SADI can also issue a form feed or any number of blank lines. The device address is switch selectible (0–15). Serial features include 11 baud rates (75 to 9600), selectible parity and a 32 character input buffer with x–on / x–off feature. For the parallel device the 'busy', 'ready' and 'data' polarities are selectible.

The SADI is easily programmed using BASIC commands, and is compatible with Wordpro, VISICALC and other software. It comes assembled and tested with case, PET IEEE cable and power supply. Thirty day money back trial period.

Retail price in USA $295.00, optional 230 V power supply $30.00.

Shirley Fletcher
Connecticut microComputer Inc.
36 Del Mar Drive
Brookfield, CT 06804
Dept 'TR'
203 775–4595
Twx: 710–456–0052

Electronic Fingerprint Analysis Security System

Identix Incorporated has recently completed two rounds of venture capital funding totaling $2.25 million. The lead investors are Citicorp Venture Capital of New York and Genesis Capital Ltd. of Bellevue, Washington.

Identix makes a computer terminal that verifies a person's identity by means of encoding a fingerprint. The terminals will be used to protect buildings and computers from unauthorized entry.

The Identix terminals are based on a patented design by the company's founder, Randall C. Fowler. Although Identix is 19 months old, the technology goes back about 12 years, when Fowler developed a device to record fingerprints on F.B.I. cards. Within the last several years, the costs of computer chips have decreased sufficiently to allow microprocessors to perform fingerprint analyses. Identix is using the Motorola 68000 chip — widely used in personal computers — in its fingerprint terminals. Production models of the terminals are being manufactured now.

Identix forsees that its terminals will be used in a number of applications: (1) to control access to buildings and laboratories and (2) for verification of persons involved in financial transactions such as automatic teller machines (ATMs). The banking and financial industries will be a major target of Identix's marketing effort.

Another large market that Identix foresees is in computer access control. An Identix terminal can positively identify a person who wishes to gain access to a computer. With the security of corporate computers being a major issue, the Identix terminals provide much greater assurance than passwords — which are the primary safeguards at this time.

In addition to company president, Randy Fowler, the management team includes: Ken Ruby, vice president of engineering; Dave Larin, vice president of marketing; and Frank Fowler, vice president of sales. Randy Fowler was formerly vice president and general manager of Flow Industries, Energy Division. Ruby was chief engineer at Motorola's Mechanical Laboratories in Phoenix, Arizona. Larin was formerly vice president of marketing at Reticon Corporation in Sunnyvale. Frank Fowler (not related to the company president) was formerly vice president of marketing at Redwood Software, San Jose.

Randy Fowler
Identix Incorporated
2452 Watson Court
Dept 'TR'
Palo Alto, CA 94303
415 858–1001

Letters

Un–products?: I am writing this letter in reference to The Transactor issues for January and July 1983. In these two issues, two products – a synthesizer keyboard and a drum synthesizer for the Commodore 64 were described. I've heard nothing of these products since. I was wondering if you had any more information as to the release of these products or even if they have been cancelled. As well, is there any other information that you might be able to send on computer music and interfacing instruments to the C64?.

R. Cooper, Thornhill, Ontario

I'm afraid both the synthesizer keyboard and the drum synthesizer have been shoved to Commodore's back burner as it were. Too bad too. Designer Paul Higginbottom, as I recall, spent many long nights working on that project and the version I saw last was literally stunning. I think Commodore's "reasoning" is that it takes 3 SID chips to make one keyboard, so for every keyboard they make, they could have made 3 C64s. Get the picture?

There are several Music packages out now for the 64 – which one is best is hard to say unless you're a musician AND a hacker. See News BRK for more info on this. Perhaps Ron Jeffries would have some advise for you. Instrument interfacing is another story. I know Chris Zamara was working on an idea, but said that problems with the SID chip (nasty clicks) had his ambition somewhat cooled. Anyone out there with further suggestions are invited to write in. – M.Ed.

Response? Response: The program listed above has become very familiar to me, chiefly through my proof–reading efforts. I have been unsuccessful in getting it to run in practise. This letter incidentally, is by a golden ager (myself) with a "B.S.W" on a C64+1541+1525 chain. Your very excellent journal lends itself admirably to the untutored neophyte in the occult art of plunking. Please be kind! I am as a village "G.P." reading about the fine points of open–heart surgery in a medical treatise. I know naught, but it is fun trying to figure it all out. Your magazine format is just great for my 3–ring binder. I could almost pun on the use of that word in this context. So, to the point; not being Jim B., I gotta ask, 'What is the correct reply to the prompt: "auto: start, increment"?'

It is probably a blunder on my part. I have been conditioned to wait a couple of months before attempting programs appearing in the media. It keeps the blood pressure down to have the expected errata in hand before transcribing a long program. It must be a gimmick to sell magazines! After the botch–up that Commodore have made of their user manuals – even I could see the goofs after a weeks trial. Closer to home, I would love to know how to put the "Function Key" thing to work!

Pardon! I am getting carried away. See you in your next edition. Keep the Ads out of the way. The "Star" would never dream of putting Mom's cross–word on Dad's sports page so you are thinking right. Sincerely,

Ralph McKnight, Hudson, Quebec

The program "above" is referring to the program "Auto Liner" that appeared in The T. Volume 4, Issue 06. However, the blunder was on our part, specifically my part. You see, in my attempts to publish versions of programs for every

Commodore model, I got somewhat hasty with Auto-Liner. The PEEK address in line 60040 of the 80 column version starts with the second space of the third line down ie. the screen start address, plus 3 times 80, plus 1, equals 33009. For the C64 version I took the screen start address (1024) and added 3 lines plus 1. But I forgot the lines are only 40 characters long. So the result was actually 6 lines, not 3. Oops. Some other errors also slipped by. Since then, a mister Keith Preston has sent us a new and improved version that we've reprinted in this issues' Bits and Pieces section.

Auto Liner's purpose in life was to reduce a little of the work involved in transcribing programs (although I admit, the first 64 version doesn't do that very well -tongue in cheek- by the way, for those reading, the 80 column version works as shown in Issue 06). It prints a line number, turns on the cursor, and enters that line for you when you hit Return. After that, the next line number is printed for you to continue. So to answer your first question, the response to "start, increment" is your choice for the first line number you wish to start you program with, followed by the amount to add to each previous line number to give you the next. For example, if you're entering a program that shows lines ascending by 10, your increment is therefore 10. "Start" is simply the first line of the program. You probably would have figured this out had the program worked - as i was, no combination of "start,increment" would have got you going.

I've used Auto Liner myself, except with one minor modification. In between the variable S and the semi-colon on line 60010, I inserted the word DATA within quotes. Now Auto Liner prints the line number, followed by "DATA" so that all I need do is enter the numbers and commas contained on each data line. Convenient. . ., when it works.

The Function Key program (I assume the one by Darren Spruyt) is a program that allows you to define what will appear when you hit a function key. This you probably know, but using it is a matter of necessity. For example, if you find yourself repeatedly PEEKing at some memory locations, you need not type PRINT PEEK(etc. every time. Just define a function key appropriately, and when you need this information, press only one key instead of retyping the whole shot. Of course you need Darren's program in order to define the function keys initially. It also works good for reading the error channel with:

```
open 1,8,15 : for j=1to40: sys43906 #1,e$
: printe$;: if st=0 then nextj
```

(Basic 2.0 use sys51844, VIC20 use sys52098) Imagine typing this every time you want to read a disk error. With Darren's utility (for the Commodore 64 only), define an F key, and depending how many disk errors you get, you'll save yourself a lot of time.

As for magazine selling gimmicks, I'd like to think that errors in listings reduce repeat sales of a magazine. I suppose errors are a fact of life, but we do try to be careful. Lastly, I agree. . . Commodore's User Manuals do leave something to be desired, but spotting "botch-ups" is an effective learning process. Their Programmers Reference Guides, though, are actually quite good. Thanks for writing and thanks for the compliments. - M.Ed.

Existing Lost Copy: I am a relatively new subscriber to your magazine. I recently received my first issue and copied several of the programs, the most recent of which was "A Simple Disk Copier For The Commodore 64" by Jim Butterfield.

In the text, Mr. Butterfield states, that when the program is run, "You'll be asked for the file type. . . Next, you'll be asked for the name of the program or file you wish to copy. . . If it can't find the file it will reply 'NO GO', otherwise it will ask OTHER DISK READY?. . . etc".

All the program does for me is copy itself on the disk, and when run agin it displays a disk error 'FILE EXISTS' and I have another useless program. Perhaps I have done something incorrectly. I admit to being a novice at this, but I am trying to learn and I find this very frustrating.

Milton Reich, Brooklyn, New York

The program you entered was actually a program generator for COPY FILE 64. (By now you have probably seen the directory that shows this program name) Once the generator program is RUN, the generator itself becomes virtually useless. . . you need not even SAVE it. A LOAD of the directory will show a new file that you yourself never put there with a SAVE command - the generator did it! The next step is:

```
LOAD "COPY FILE 64" ,8
```

When that finishes, you can LIST COPY FILE 64 and see the program that was "generated". RUN this program and follow the original instructions. Perhaps this was a little unclear in the article. Just remember that a program "generator" is a program that writes another program which must be subsequently LOADed to be used. After this is done once, the generator is of no further use, but the program you have now, "COPY FILE 64", will be of much further use I'm sure- M.Ed.

Bits and Pieces

Kernal 3 For The Commodore 64

Commodore has released Kernal 3 – a new retro fit ROM for the C64. The "Kernal" is one of 4 ROMs found inside the 64. It's called the Kernal because it handles the fundamental or "inner most" operations of the machine. Reportedly, fixes over Kernal 2 are:

1) The INPUT command has been fixed so that the INPUT prompt is not included with the response when the prompt is greater than 40 characters.

2) The problem with DELeting the last character of the last line on the screen has been corrected. Recall, if you start typing on the last line of the screen for 80 characters such that the screen scrolls twice, and then use DEL to move back and delete the 80th character, the CIA that lies above the colour table is disturbed and becomes very unfriendly. Now eliminated.

3) A problem was found in the RS–232 routines that occurred with either even or odd parity enabled that could result in inaccurate status reads.

4) Serial Bus Timing has been slightly modified to allow for several chained peripherals. When too many peripherals were connected on the serial bus the system would occasionally misbehave.

To test for Kernal 3, PRINT PEEK(65408). Details of price and availability are not yet available – call your local dealer or Commodore Service.

Cylinder Screen

For an interesting but useless screen effect on your 8000/9000 series machine, try this POKE from Dave Gzik of Burlington, Ontario:

```
POKE 59521, 40
```

When the video chip recovers from this punch you'll notice that your screen has bee twisted into a cylinder. Reset or PRINT CHR$(14) will restore order.

Down Scroll 64

Another of Murphy's unwritten laws states that "while trying to accomplish a specific task you will always accomplish some other task that brings you no closer to your original goal". Paul Blair of Holder, Australia has reconfirmed this law with the following submission.

". . .came across this while doing something else – all the best discoveries happen that way. The routine will scroll the Commodore 64 screen down starting from line D ie. from the top line with D=0, second line with D=1, etc. Colour changes from line to line are also allowed. At the end of the routine, some pointers are left a bit untidy, so use with caution. A PRINT or two on the end seems to restore order. . . thought you might like it – regards, Paul Blair".

```
100 d=0 : x=211 : v=15 : a=53280
110 poke a, 1 : poke a+1, 3
120 print" ■ ";
130 read a$ : v=v-1 : poke a, v : if a$=" end " then end
140 print" ■ ";
150 for t=1 to 10 : print a$;d : next
160 for t=0 to 14 : poke x+3, d : sys 59749 : next
170 print : for dl=1 to 2000 : next : d=d+1 : goto 130
180 data " scroll down with this pgm "
190 data " it's really very easy to use "
200 data " include it in games and so on "
210 data " list the pgm to see the set up "
220 data " see how you can select scroll start? "
230 data " have fun . . . . . paul blair "
240 data " end "
```

Equivalent VIC 20 and BASIC 2.0 routines have not been investigated but presumably would work depending on their ROMs. Fat 40 and 8000/9000 series machines don't need a routine like this – use PRINT CHR$(153) instead.

FTOUTSM With Colour Mods

Remember FTOUTSM? – For Those Of Us That Smoke M——? Originally written by Benny Pruden of Norristown, PA, it has since been updated by Louis Black of Oshawa, Ontario to include colour on the C64. A VIC 20 version would not pose too big a problem. . . just swap out the numbers that reflect the screen width and address locations, as well as the POKEs in line 4 for border and background colours, and swap in the appropriate VIC 20 equivalents. Line 1 and 3 must be entered using abbreviated keywords on at least some of the commands to make them fit on one line. If abbreviations are new to you, see Louis Sanders' article this issue on Commodore BASIC Abbreviations.

```
0 print" ■ ";
1 c=32:forn=1to41:gosub3:c=192-c:fora=0ton
  :forb=1024+ato2024stepn:pokeb,c:nextb,a,n
2 end
3 x=int(15*rnd(1)):y=int(15*rnd(1)):poke53280,x
  :poke53281,y:fori=1to100:next:return
```

Machine Language FTOUTSM

In keeping with our theme this issue, here's FTOUTSM in machine language for the 64. Writer Chris Zamara said he had to insert a delay loop into the code because it was just too fast to have any adverse effect on your brain. Although it's still faster than the BASIC version, you will also notice that it's much smoother. Once again, the Surgeon General advises that danger to mental health increases geometrically with the number of FTOUTSM iterations. And as they say on the 20–Minute Workout, "do not over FTOUTSM yourself". And Murphys' first law says, "if something can go FTOUTSM, it will". And Mr T. says "jus try it, fool"

```
1000 rem machine code ftoutsm
1010 for j=49152 to 49330 : read x
1020 poke j, x : ch=ch+x : next
1030 if ch <> 24671  then print" checksum error " : end
1040 sys 49152 : goto 1040
1050 data  76,   9, 192,   0,   0,  32,   0,   9
1060 data  50, 160,   0,  32, 129, 192, 169,  32
1070 data 141,   5, 192, 169,   1, 141,   6, 192
1080 data  32,  52, 192,  32, 129, 192, 165, 197
1090 data 201,  63, 240,  10, 238,   6, 192, 173
1100 data   6, 192, 201,  42, 144, 234, 169,   0
1110 data 141,  33, 208,  96, 173,   5, 192,  73
1120 data 128, 141,   5, 192, 169,   0, 141,   3
1130 data 192,  32,  82, 192, 238,   3, 192, 173
1140 data   3, 192, 205,   6, 192, 240, 242, 144
1150 data 240,  96,  24, 173,   3, 192, 105,   0
1160 data 133, 253, 169,   0, 105,   4, 133, 254
1170 data 173,   5, 192, 145, 253,  32, 169, 192
1180 data  24, 165, 253, 109,   6, 192, 133, 253
1190 data 165, 254, 105,   0, 133, 254, 201,   7
1200 data 144, 230, 165, 253, 201, 192, 144, 224
1210 data  96, 169,   0, 133, 251, 169, 216, 133
1220 data 252, 173,   7, 192, 145, 251, 230, 251
1230 data 208,   2, 230, 252, 165, 252, 201, 219
1240 data 144, 239, 165, 251, 201, 232, 144, 233
1250 data 173,  33, 208, 205,   7, 192, 208,   0
1260 data  96, 174,   8, 192, 234, 234, 234, 202
1270 data 208, 250,  96
```

amaZAMARAing

(Sorry Chris – I just couldn't resist it) Here's another blitzoid screenzler: Timescroll for the C64 from Chris Zamara of Downsview, Ontario. Notice how the line is padded with spaces in two spots? Change the number of these spaces for different effects. Line 20 details the exact number to start with. You can also change variable R to 53280 (the border colour register) for madded adness.

```
10 a=0:b=1:r=53281:fori=0to1step0
   :   poker,a:      poker,b:next
20 rem step0:3 spaces poker,a: 7 spaces poke etc.
```

Quick Note: The VIC 20, matched task for task, is the fastest of the Commodore machines.

Stop RUN/STOP

Most of you have no doubt seen at least one RUN/STOP disable for the C64. The following POKE was published several issues ago. It disables RUN/STOP (and RUN/STOP-RESTORE) without affecting the TI clock, but don't try LOADing or SAVing and expect normal results!

```
POKE 808, PEEK(808)-16
```

Therefore, this should only be used after the program has been LOADed and only with program that do not LOAD subsequent software modules. This next routine is by James Whitewood of Milton, Ontario. It does everything the above POKE does without messing up LOAD and SAVE:

```
10 lo = 12 * 4096
20 c = int(lo/256) : b = lo-c*256
30 for i = lo to i + 4 : read a : poke i,a : next
40 poke 808, b : poke 809, c : end
50 data 169, 255, 133, 145, 96
```

The address computed in line 10 as variable LO can be any available memory ie. the cassette buffer will host this routine just fine. Notice how line 30 uses the loop variable I in the calculation "I+4" to specify the end of the loop. This is quite legal since I is set to LO and entered in the simple variables table (just like any other variable) before BASIC interprets the TO operative. However, you might also notice that 4 is one less than the number of DATA items. In situations like these, inclusive logic must be used to determine the number of loop iterations.

Cursed Commodore Cursor!

Keith Preston of Ottawa, Ontario, has these comments on invoking the built-in cursor routines while a program is running, as detailed in The T.

"Several articles in Volume 4, Issue 6 suggest that a flashing cursor, the neophyte's comforter, may be retained during a Commodore GET by invoking POKE 204, 0. These are "Auto Liner" on page 18, "Subroutine Eliminators" on page 37 and "Three GET Subroutines" on page 38. When using the C64, however, the single POKE does not guarantee a flashing cursor for more than the first character of an input string (as requested in "Auto Liner"). Furthermore, the cursor may disappear upon hitting RETURN! To prevent this, simply add:

```
POKE 207, 0
```

in any line after the GET. A further:

```
POKE 204, 1 : POKE 207, 0
```

before exiting the input routine ensures a return to normal cursor function.

The accompanying short routine illustrates the technique and should be used to replace "Auto Liner". A number of other minor errors in that program have also been corrected."

```
60000 input "64 auto: start, increment";s,i
60010 print " Sqqq "; s;:poke204,0
60020 geta$ : if a$ = "" then 60020
60030 poke 207, 0 : print a$; : if asc(a$)<>13
      then 60020
60040 p = peek(1145 + len(str$(s))) : if p = 32 or p = 160
      then 60010
60050 print "s = "s+i ":i = "i ":goto60010 s "
60060 poke 631,13 : poke632,13 : poke198, 2
60070 poke 204, 1 : poke 207, 0 : end
```

```
60000 input "4.0/2.0 auto: start, increment";s,i
60010 print " Sqqq "; s;:poke167,0
60020 get a$ : if a$ = "" then 60020
60030 poke 170, 0 : print a$; : if asc(a$)<>13
      then 60020
60040 p = peek(33009 + len(str$(s))) : if p = 32 or p = 160
      then 60010
60050 print "s = "s+i ":i = "i ":goto60010 s "
60060 poke 623,13 : poke624,13 : poke 158, 2
60070 poke 167, 1 : poke 170, 0 : end
```

Also have a look at Elizabeth Deal's article, "To GET Or Not To GET", later in this issue – Ed.

Sorry, But That DOES Compute

Ernest Blaschke of Sudbury, Ontario has these comments:

"In the commercial world, we all have heard the phrase: "Sorry, the computer made a mistake!". We know, of course, that it is the programmer and not the computer that made the mistake. Computers don't make mistakes. Right?

Well let me show you that your computer will make mistakes and will logically contradict itself. Yet, not all is lost. A programmer should know the computers' weaknesses and keep it from making true mistakes.

Type into your computer the direct command:

```
PRINT 5 ↑ 8
```

The reply will be 390625. The computer has in fact produced the correct value which is 5*5*5*5*5*5*5*5

Now enter the following small program:

```
10 if 5↑8  = 390625 then print "true"
20 if 5↑8 <> 390625 then print "false"
```

Type "RUN" and the computer will print "false", contradicting its previous statement that 5↑8=390625.

You probably know that your computer will reply with -1 to a true statement and with 0 to a false one.

If you aren't sure about this, try:

PRINT (2*2=4)

The computer replies with -1 (true). PRINT (2*2=5) will result in 0 (false). However, even using this approach, the computer stubbornly denies its own findings that 5↑8=390625.

PRINT (5↑8=390625) will reply with 0, false.

So what happened? The problem is that the computer calculates 5↑8 in floating point arithmetic and due to roundoff errors thinks the result is slightly greater than 390625. For printing, it "rounds off" the value in memory to the correct 390625. However, equality tests fail since the computer perceives the true result to be larger, and therefore unequal.

There are whole sets of problems where it is essential for the programmer to avoid this pitfall in order for the computer to do its task reliably. Bearing potential roundoff errors in mind, the programmer should have typed:

PRINT (INT(5↑8)=390625)

This would result with the -1 or true response. Of course this is limited to numbers that can be anticipated to have no fractional content. For numbers with magnitude to the right of the decimal point, the programmer should consider moving the decimal point right by multiplying by some multiple of 10, say 100 or 1000, or as many significant digits as desired. Then take the INTeger portion of this number and divide by the same multiple of 10.

I hope to have convinced you may not blindly trust everything that appears on the screen or consider your computer's logic infalible."

Low-Res Screen Copy

If you've ever attempted to do a low resolution screen dump of a screen containing graphics, you've seen that the printer leaves a little horrible space on carriage returns. This leaves the printout looking like it went through a shredder. But by using "LOW RES COPY", you can eliminate that space on the printout. The program itself is only 14 lines long, somewhat shorter than the 22 lines of "Screen Copy" in the VIC 1525 user's manual. I find this program to be a very handy utility when the time arises that you need a true low resolution screen copy. Brian Dobbs, Timmins, Ontario.

```
100 si$=chr$(15) : bs$=chr$(8) : d=1024 : open4,4
110 for a=d to d+39
120 print#4, si$;
130 b=peek(a)
140 if b>-1 and b<32 then e$=chr$(b+64)
150 if b>31 and b<64 then e$=chr$(b)
160 if b>63 and b<96 then e$=chr$(b+32)
170 if b>95 and b<128 then e$=chr$(b+64)
180 print#4, e$;
190 next
200 print#4, bs$
210 d=d+40 : if d>1984 then 230
220 goto 110
230 end
```

Eep Eep

Eep Eep is a short interrupt driven routine that uses the cursor countdown timing register to drive the CB2 transducer (it's not really a speaker so it's called a transducer). Eep Eep only works on BASIC 4.0 machines but could be modified to drive the SID or VIC 20 sound registers. However, it's only good for two things really: one, it demonstrates the concept of pre-interrupt code. Notice the first 9 numbers in the DATA statements - you can almost read them without a dissassembler. They go LDA with 131, STA in location 144, LDA with 2, STA in location 145, and RTS (96). 2 times 256 plus 131 equals 643 which is where the actual pre-interrupt program begins (LDA with 16 right after the RTS). This is one of the most common methods to engage a pre-interrupt routine, and the quickest ways to spot one - something to remember when you find some old listing lying around.

At the end of line 1090 are three 234's. These are NOPs. It means simply No OPeration or NO oPeration, whichever you prefer. The reason these are here is to accommodate the three POKEs in line 1035. Line 1035 can be left out for a different Eep Eep. RUN the program as is, then remove 1035 and RUN again.

Line 1100 contains the code JMP to location $E455. This is the regular interrupt routine that the computer usually goes to when there is no pre–interrupt code – another way to spot pre–interrupt routines.

Eep Eep plays with the same chip responsible for LOADs and SAVEs. It's suggested you purge your machine of Eep Eep before continuing with more serious work.

Oh ya, the other thing Eep Eep does effectively is drive you bonkers. Just hook your computer up to your stereo, start Eep Eep, and tell no–one to touch your equipment. Then leave.

```
1000 rem eep eep – rte 1984
1010 for j=634 to 676 : read x
1020 poke j, x : ch=ch+x : next
1030 if ch <> 6145  then print "checksum error"  : end
1035 poke 671, 238 : poke 672, 147 : poke 673, 2
1040 sys 634
1050 data 169, 131, 133, 144, 169,   2, 133, 145
1060 data  96, 169,  16, 141,  75, 232, 169,  20
1070 data 141,  74, 232, 165, 168, 141,  72, 232
1080 data 160,   0, 200, 208, 253, 169,   0, 141
1090 data  75, 232, 141,  74, 232, 234, 234, 234
1100 data  76,  85, 228
```

Mirror

Mirror is another pre–interrupt routine also written by Richard Evers. It was written for no other reason but to see it work.

```
1000 rem mirror 40 – rte 1984
1010 for j=634 to 682 : read x
1020 poke j, x : ch=ch+x : next
1030 if ch <> 6710  then print "checksum error"  : end
1040 sys 634
1050 data 169, 131, 133, 144, 169,   2, 133, 145
1060 data  96, 162,   0, 160, 255, 189,   0, 128
1070 data 153, 232, 130, 136, 232, 208, 246, 238
1080 data 137,   2, 206, 140,   2, 173, 137,   2
1090 data 201, 130, 208, 233, 169, 128, 141, 137
1100 data   2, 169, 130, 141, 140,   2,  76,  85
1110 data 228
```

```
1000 rem mirror 80 – rte 1984
1010 for j=634 to 682 : read x
1020 poke j, x : ch=ch+x : next
1030 if ch <> 6696  then print "checksum error"  : end
1040 sys 634
1050 data 169, 131, 133, 144, 169,   2, 133, 145
1060 data  96, 162,   0, 160, 255, 189,   0, 128
1070 data 153, 208, 134, 136, 232, 208, 246, 238
1080 data 137,   2, 206, 140,   2, 173, 137,   2
1090 data 201, 132, 208, 233, 169, 128, 141, 137
1100 data   2, 169, 134, 141, 140,   2,  76,  85
1110 data 228
```

The C64 version is a little longer due to colour table servicing required for Kernal 2 machines. However, it stops working after a Clear Screen is done, until the POKE in line 1040 is given. Can someone help us here? It's probably just some silly oversight that we can't seem to spot because of the clouds between us and the screen – you know the ones we mean, they're made of clear air? Hmm.

```
1000 rem mirror 64 – rte 1984
1010 for j=828 to 900 : read x
1020 poke j, x : ch=ch+x : next
1030 if ch<> 8190  then print "checksum error"  : end
1040 poke 53281, 493–peek(53281) : sys 828
1050 data 169,  71, 141,  20,   3, 169,   3, 141
1060 data  21,   3,  96, 162,   0, 160, 255, 189
1070 data   0,   4, 153, 232,   6, 189,   0, 184
1080 data 153, 232, 186, 136, 232, 208, 240, 238
1090 data  77,   3, 206,  80,   3, 238,  83,   3
1100 data 206,  86,   3, 173,  77,   3, 201,   6
1110 data 208, 221, 169,   4, 141,  77,   3, 169
1120 data   6, 141,  80,   3, 169, 184, 141,  83
1130 data   3, 169, 186, 141,  86,   3,  76,  49
1140 data 234
```

Ram Scan

Ram Scan might be useful to somebody out there. Once engaged, it continually displays as many bytes of memory as will fit on the screen. Positioned over Zero Page, it will show the various timers, etc, in action. Same with the VIA and PIA registers up at $E800. To move the display use the cursor keys – cursor up/down moves it by one line of bytes, cursor left/right by one byte at a time. The STOP key puts you back in BASIC. Other than this, it too will give some pretty eye crossing patterns, something Richard seems to enjoy inflicting. Try moving the display around just below, and then above the first screen address.

```
1000 rem ram scan 80 – rte 1984
1010 for j=634 to 724 : read x
1020 poke j, x : ch=ch+x : next
1030 if ch <> 11974  then print "checksum error"  : end
1040 sys 634
```

```
1050 data 165, 151, 201, 255, 240,  43, 166, 152
1060 data 224,   0, 208,  10, 201,  17, 208,  16
1070 data 238, 178,   2,  76, 171,   2, 201,  17
1080 data 208,  16, 206, 178,   2,  76, 171,   2
1090 data 201,  29, 208,   6, 238, 177,   2,  76
1100 data 171,   2, 201,  29, 208,   3, 206, 177
1110 data   2, 160,   0, 174, 178,   2, 185,   0
1120 data 255, 153,   0, 128, 200, 208, 247, 238
1130 data 178,   2, 238, 181,   2, 173, 181,   2
1140 data 201, 136, 208, 234, 142, 178,   2, 169
1150 data 128, 141, 181,   2, 165, 155, 201, 239
1160 data 208, 166,  96
```

```
1000 rem ram scan 40
1010 for j=634 to 744 : read x
1020 poke j, x : ch=ch+x : next
1030 if ch <> 14739  then print " checksum error "  : end
1040 sys 634
1050 data 169, 147,  32, 210, 255, 165, 151, 201
1060 data 255, 240,  41, 166, 152, 208,  10, 201
1070 data  17, 208,  16, 238, 199,   2,  76, 174
1080 data   2, 201,  17, 208,  16, 206, 199,   2
1090 data  76, 174,   2, 201,  29, 208,   6, 238
1100 data 198,   2,  76, 174,   2, 201,  29, 208
1110 data   3, 206, 198,   2, 173, 198,   2, 133
1120 data 251, 173, 199,   2, 133, 252, 169,  19
1130 data  32, 210, 255,  32,  23, 215, 160,   0
1140 data 174, 199,   2, 185,   0, 255, 153,   5
1150 data 128, 200, 208, 247, 238, 199,   2, 238
1160 data 202,   2, 173, 202,   2, 201, 132, 208
1170 data 234, 142, 199,   2, 169, 128, 141, 202
1180 data   2,  32, 225, 255,  76, 127,   2
```

```
1000 rem ram scan 64
1010 for j=828 to 916 : read x
1020 poke j, x : ch=ch+x : next
1030 if ch <> 10348  then print " checksum error "  : end
1040 poke 53281, 493-peek(53281) : sys 828
1050 data 165, 203, 201,  64, 240,  42, 174, 141
1060 data   2, 208,  10, 201,   7, 208,  16, 238
1070 data 115,   3, 184,  80,  27, 201,   7, 208
1080 data  16, 206, 115,   3, 184,  80,  17, 201
1090 data   2, 208,   6, 238, 114,   3, 184,  80
1100 data   7, 201,   2, 208,   3, 206, 114,   3
1110 data 160,   0, 174, 115,   3, 185,   0, 255
1120 data 153,   0,   4, 200, 208, 247, 238, 115
1130 data   3, 238, 118,   3, 173, 118,   3, 201
1140 data   8, 208, 234, 142, 115,   3, 169,   4
1150 data 141, 118,   3,  32, 225, 255, 184,  80
1160 data 167
```

Crystal

Crystal is just a short little program that draws a crystaline pattern on your screen. Aside from that it demos how very little code it takes to get something happening – something like a game layout, a game intro, or an attract mode for a game you may have just finished and thought you didn't have room for an attract mode feature.

Crystal also demonstrates a technique that all programmers should be used to or else get used to – portability. Some programs aren't suited to be run on all machines, but those that could potentially be run on any machine should include for the user all necessary conversion information. It doesn't take long and it's a courtesy that adds an extra professional touch.

```
100 rem crystal
110 rem 8000/9000 series  : sw=80
120 rem 4000 + c64        : sw=40
130 rem vic 20            : sw=22
140 rem 4.0 basic         : ss=32768
150 rem c64               : ss=1024 (default)
160 rem vic 20            : ss=7680 (default)
170 rem sw=screen width : ss=screen start
180 print " SN " ;: ss=32768 : sw=80
    : rem * place your variables here
190 x=1:y=1:dx=1:dy=1
200 poke ss + x + sw*y,81 : poke ss + x + sw*y,91
210 x = x + dx : if x = 0 or x = sw-1 then dx = -dx
220 y = y + dy : if y = 0 or y = 24   then dy = -dy
230 s=peek(ss+x+sw*y) : ifs=91thendx=-dx
    : poke ss+x+sw*y, 86 : goto210
240 goto 200
```

Number Base Converter

This next program works on BASIC 4.0 machines only because it uses some internal ROM routines of the built in Machine Language Monitor which the other machines don't have. Quite simply, it will convert numbers from one number base to another that are in hexadecimal, decimal, or binary.

There are two internal ROM routines used here: the first, SYS HD (where HD=55124), inputs a hexadecimal number from the keyboard and places its high order and low order components in locations 252 and 251. The program takes over from there and uses variable NO to build a decimal representation (line 12 or 13).

The second, SYS DH (where DH=55063), is the MLM routine for outputing a hexadecimal number whose high order and low order components are in locations 252 and 251 (line 15 or 19).

```
0 rem save"@0:hex/dec/bin conv",8:verify
  "0:hex/dec/bin conv",8
1 rem * richard evers - march 8th 1984 - 4.0 only *
10 input"[ st]h[R]ex>dec, hex>[r]b[R]in, [r]d[R]ec>hex,
   dec>[r]B[R]in, bin>[r]H[R]ex, bin>[r]D[R]ec";q$
11 print"[ S]";:hd=55124:dh=55063:if q$="B"then
   input"[u]decimal ";no:goto16
12 if q$="b"then print"[u]hex val ";:syshd
   :no=peek(251)+256*peek(252):goto16
13 if q$="h"then print"[u]hex val ";:syshd
   :printpeek(251)+256*peek(252):goto10
14 if q$="H" or q$="D" then input"[u]binary
   number";bn$ : goto17
15 input"[u]decimal ";a:b=int(a/256):c=a-256*b
   :poke251,c:poke252,b:sysdh:goto10
16 print:a=32768:forc=1to16:b=int(no/a):printb;
   :no=no-b*a:a=a/2:nextc:goto10
17 a=0:c=1:forb=len(bn$)to1step-1
   :a=a+val(mid$(bn$,b,1))*c:c=c*2:nextb
18 ifq$="D"then print"decimal"a : goto10
19 print"$";:b=int(a/256):c=a-256*b
   :poke251,c:poke252,b:sysdh:goto10
```

The Un-Cursor

Still another pre-interrupt routine is this one called Un-Cursor. As the name might imply, Un-Cursor flashes everything on the screen except the space at the cursor position. At least that was the original intention - the real cursor seems to slip in an appearance every once in a while.

These pre-interrupt routines we've been bombarding you with may have no place in your utilities library, but they do serve one vital purpose. By giving you several examples we believe we accomplish two things - eliminating the fear and apprehension of messing with the fundamental operation of the machine is an important step towards becoming proficient with your computer. And second, when you come up with your own idea for a pre-interrupt program, we hope one of these examples will serve as a guide to completing your task.

```
1000 rem un-cursor 80
1010 for j=634 to 692 : read x
1020 poke j, x : ch=ch+x : next
1030 if ch <> 7656  then print"checksum error" : end
1040 sys 634
1050 data 169, 131, 133, 144, 169,   2, 133, 145
1060 data  96, 165, 170, 201,   1, 240,  41, 169
1070 data 128, 133,  88, 169,   0, 133,  87, 168
1080 data 177,  87,  73, 128, 145,  87, 200, 208
1090 data 247, 230,  88, 165,  88, 201, 136, 208
1100 data 239, 238, 134,   2, 173, 134,   2, 201
1110 data   2, 208,   5, 169,   0, 141, 134,   2
1120 data  76,  85, 228
```

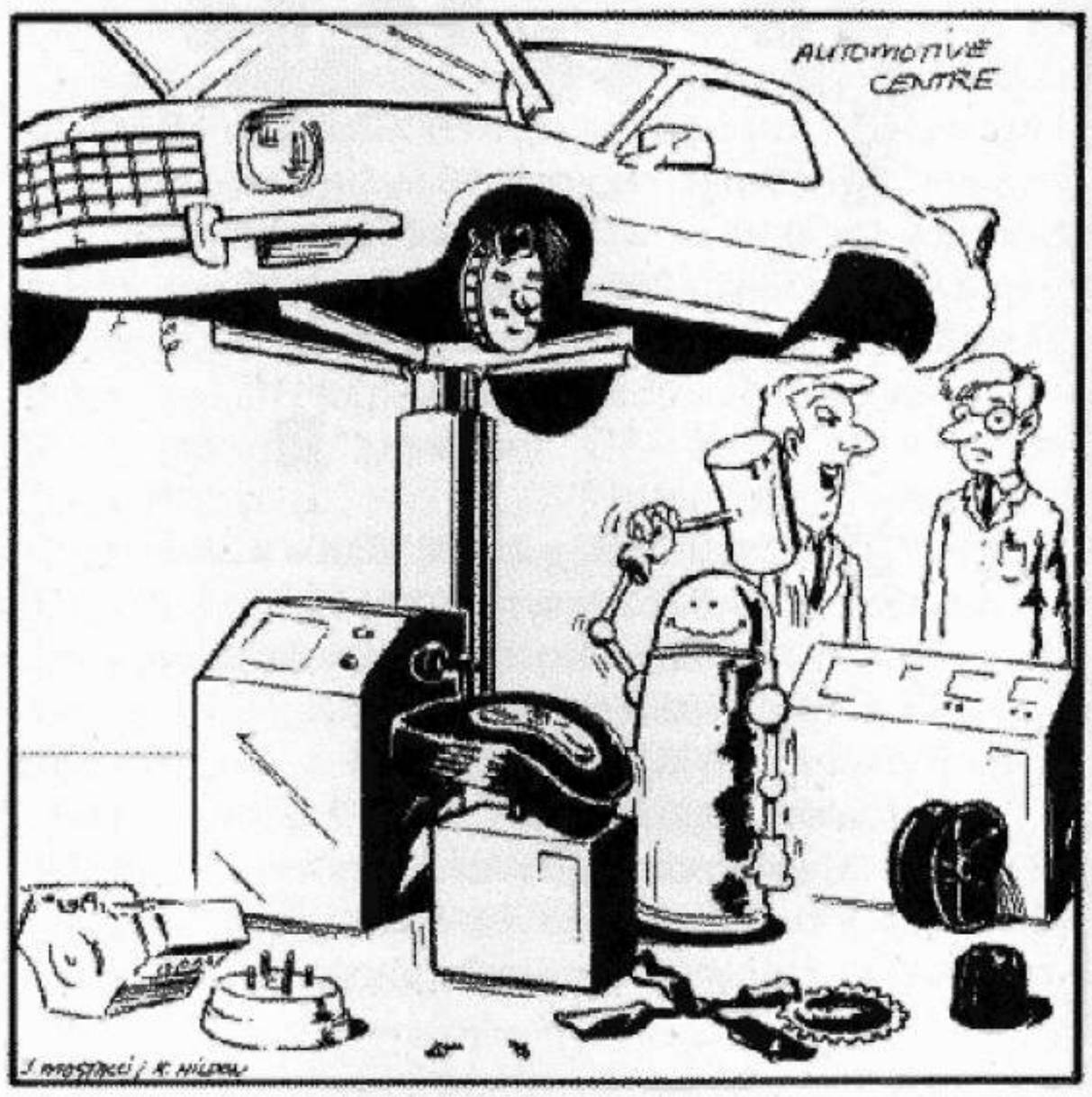
```
1000 rem un-cursor 40
1010 for j=634 to 692 : read x
1020 poke j, x : ch=ch+x : next
1030 if ch <> 7652  then print"checksum error" : end
1040 sys 634
1050 data 169, 131, 133, 144, 169,   2, 133, 145
1060 data  96, 165, 170, 201,   1, 240,  41, 169
1070 data 128, 133,  88, 169,   0, 133,  87, 168
1080 data 177,  87,  73, 128, 145,  87, 200, 208
1090 data 247, 230,  88, 165,  88, 201, 132, 208
1100 data 239, 238, 134,   2, 173, 134,   2, 201
1110 data   2, 208,   5, 169,   0, 141, 134,   2
1120 data  76,  85, 228
```

```
1000 rem un-cursor 64
1010 for j=828 to 888 : read x
1020 poke j, x : ch=ch+x : next
1030 if ch <> 6949  then print"checksum error" : end
1040 sys 828
1050 data 169,  71, 141,  20,   3, 169,   3, 141
1060 data  21,   3,  96, 165, 207, 201,   1, 240
1070 data  41, 169,   4, 133,  88, 169,   0, 133
1080 data  87, 168, 177,  87,  73, 128, 145,  87
1090 data 200, 208, 247, 230,  88, 165,  88, 201
1100 data   8, 208, 239, 238,  74,   3, 173,  74
1110 data   3, 201,   2, 208,   5, 169,   0, 141
1120 data  74,   3,  76,  49, 234
```

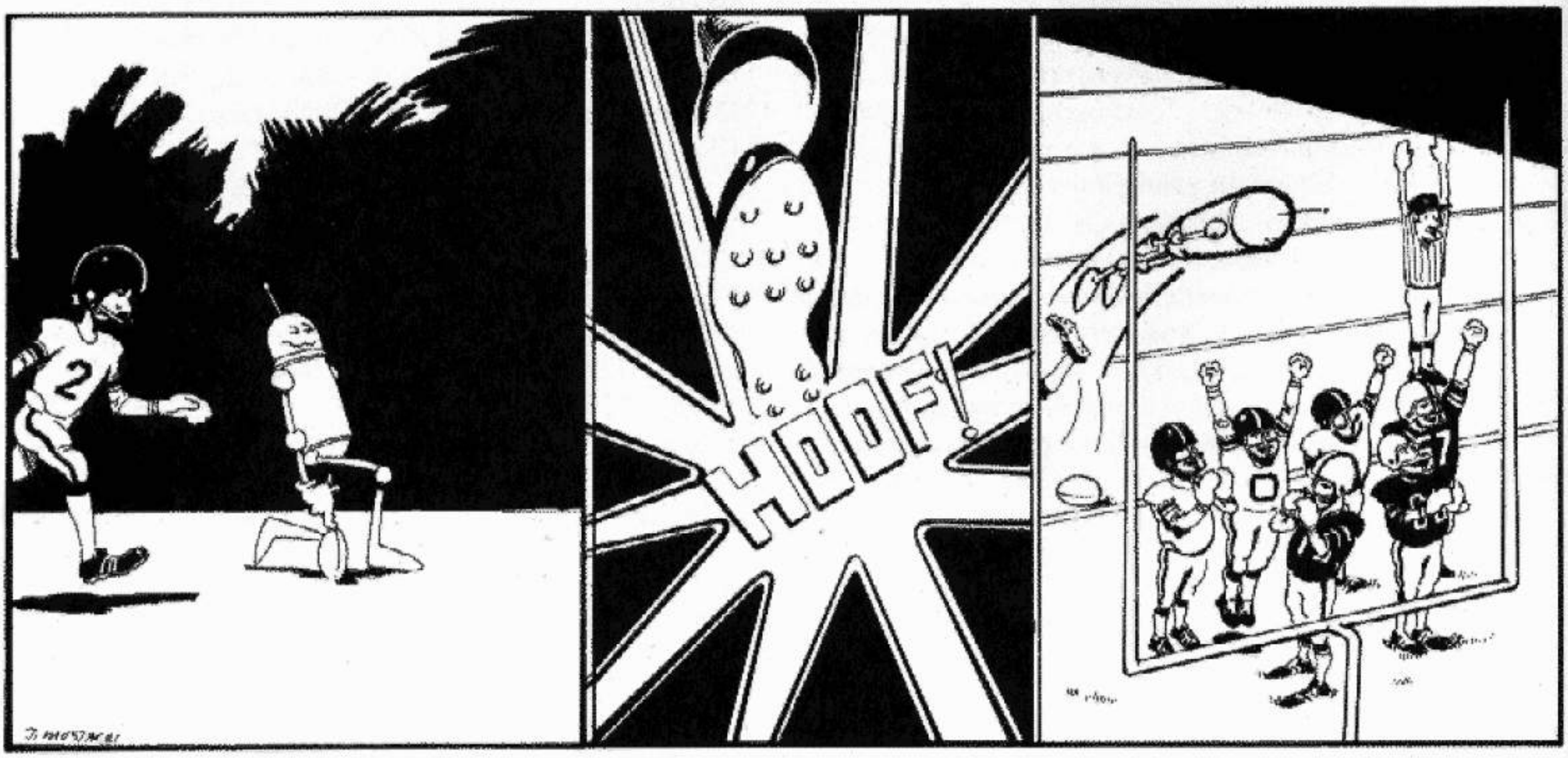
```
1000 rem un-cursor 20
1010 for j= 828 to 888 : read x
1020 poke j, x : ch=ch+x : next
1030 if ch <> 7141  then print"checksum error" : end
1040 sys 828
1050 data 169,  71, 141,  20,   3, 169,   3, 141
1060 data  21,   3,  96, 165, 207, 201,   1, 240
1070 data  41, 169,  30, 133,  88, 169,   0, 133
1080 data  87, 168, 177,  87,  73, 128, 145,  87
1090 data 200, 208, 247, 230,  88, 165,  88, 201
1100 data  32, 208, 239, 238,  74,   3, 173,  74
1110 data   3, 201,   2, 208,   5, 169,   0, 141
1120 data  74,   3,  76, 191, 234
```

CompuKinks.

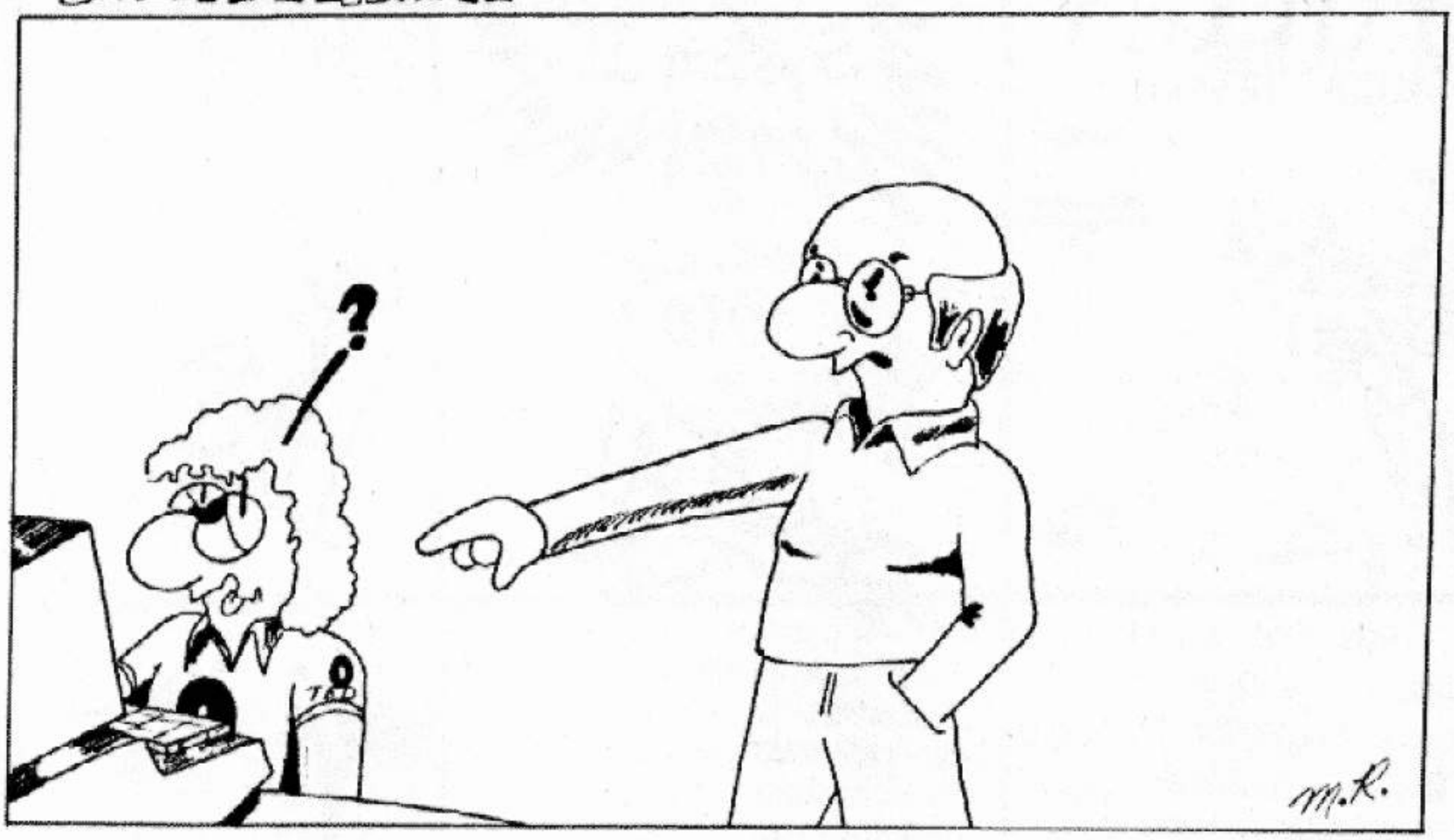
All right Earthling, turn around slowly
and keep your hands up!



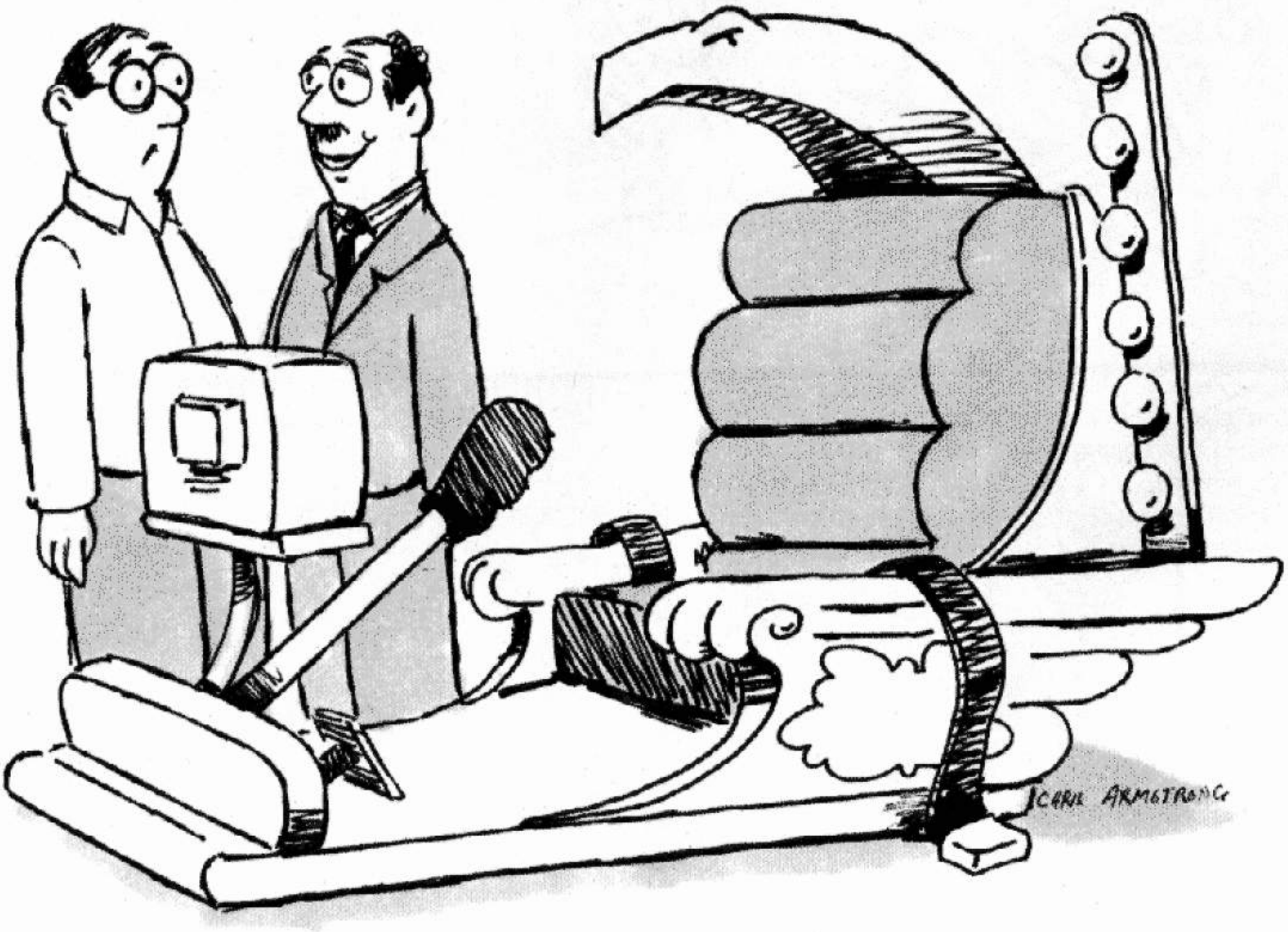
He's the new shop assistant.
He seems to like it here,
but I don't think he's gonna work out.



DATA-MAMA! STARRING TED PETROSKY



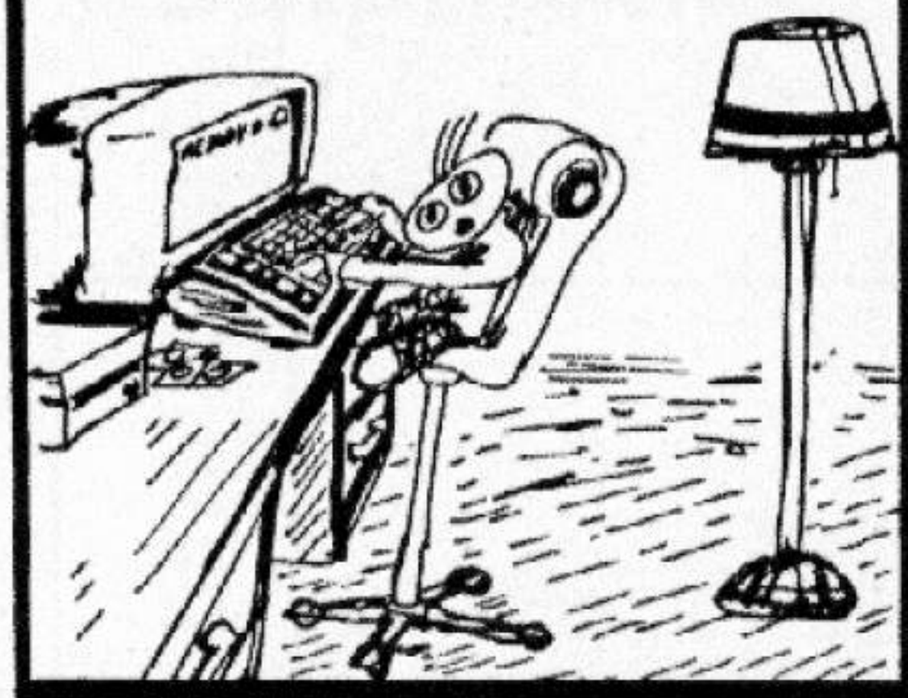
"NO TED, THERE'S NO SUCH THING AS A WHAT...FOR STATEMENT!"

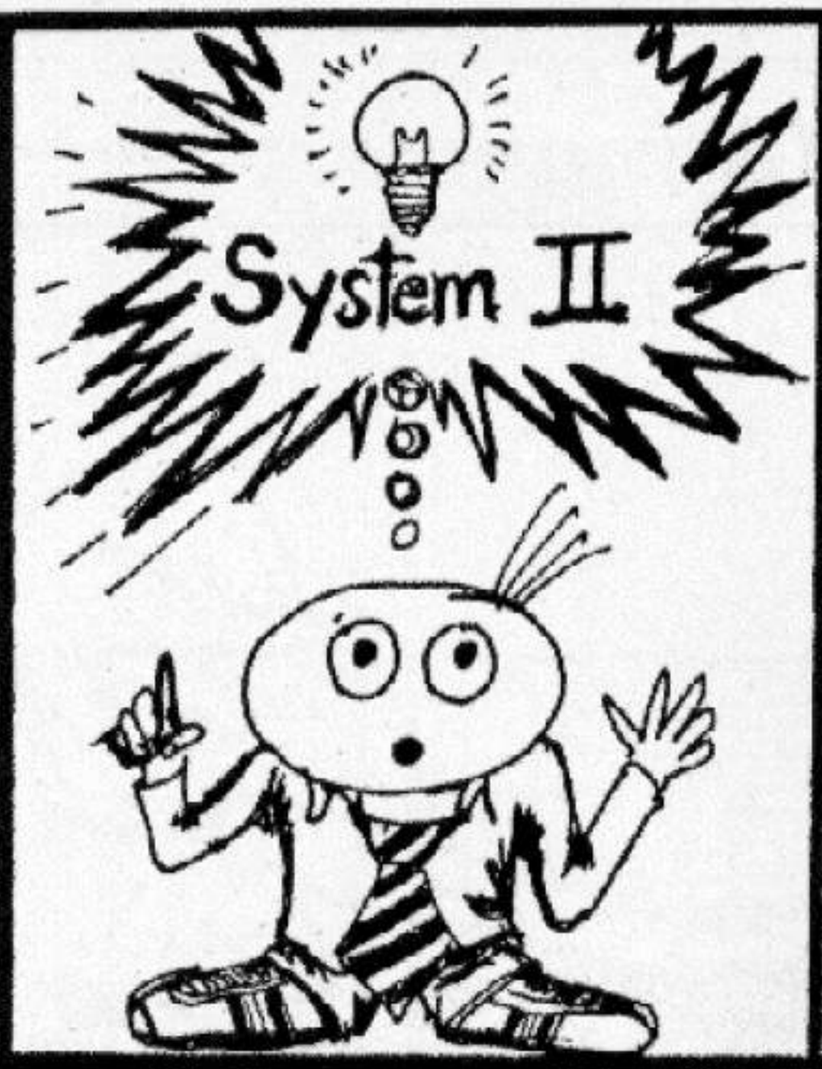


THIS IS OUR NEW DELUXE JOYSTICK

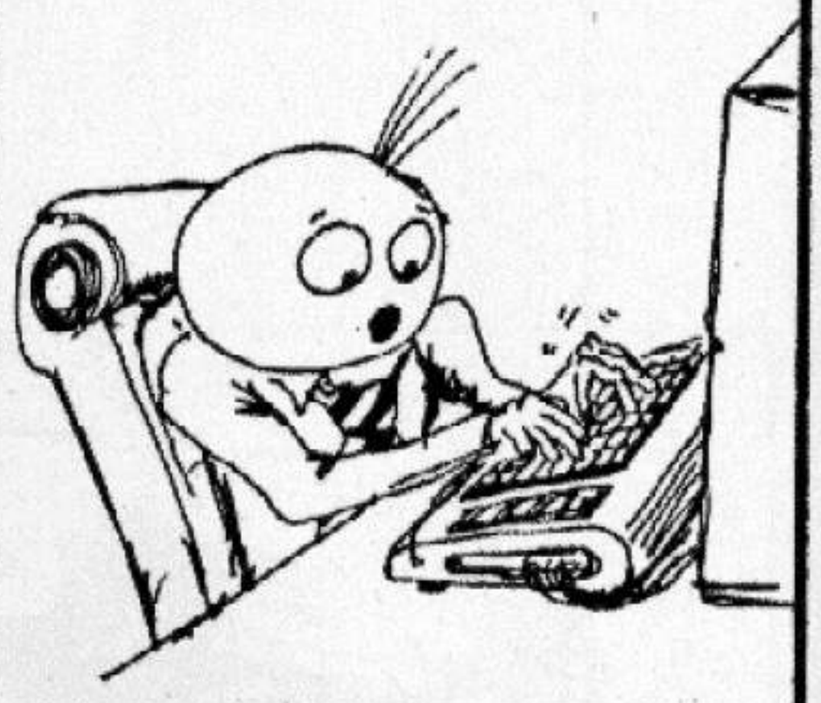
ARWUN

COMPUTER GENIUS

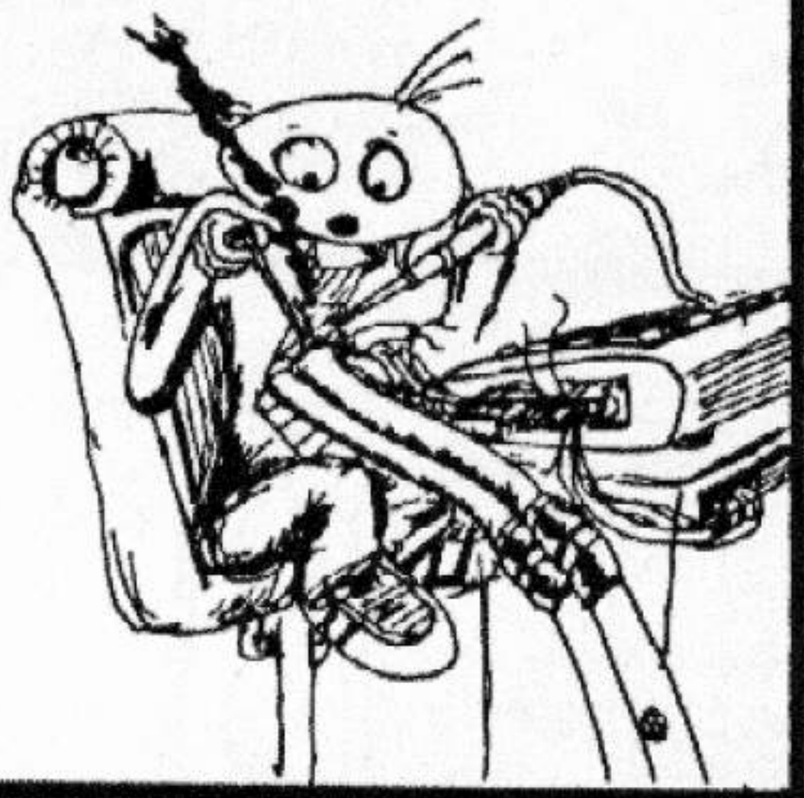




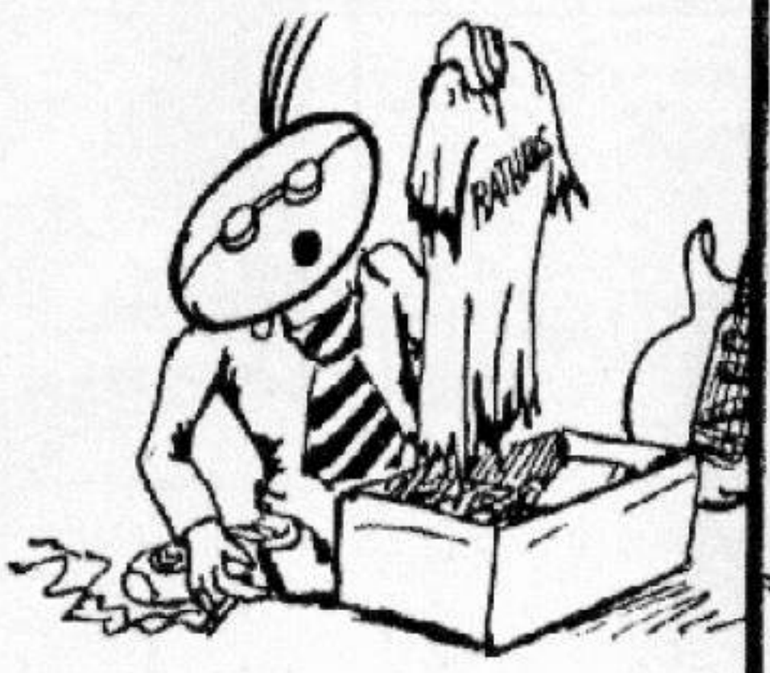
STEP I -THE "SID" CHIP MUST BE SET UP FOR PROPER ATTACK, DECAY, SUSTAIN, RELEASE, FILTRATION AND FREQUENCY CHARACTERISTICS. (ARWUN IS ENTHUSIASTIC.....)

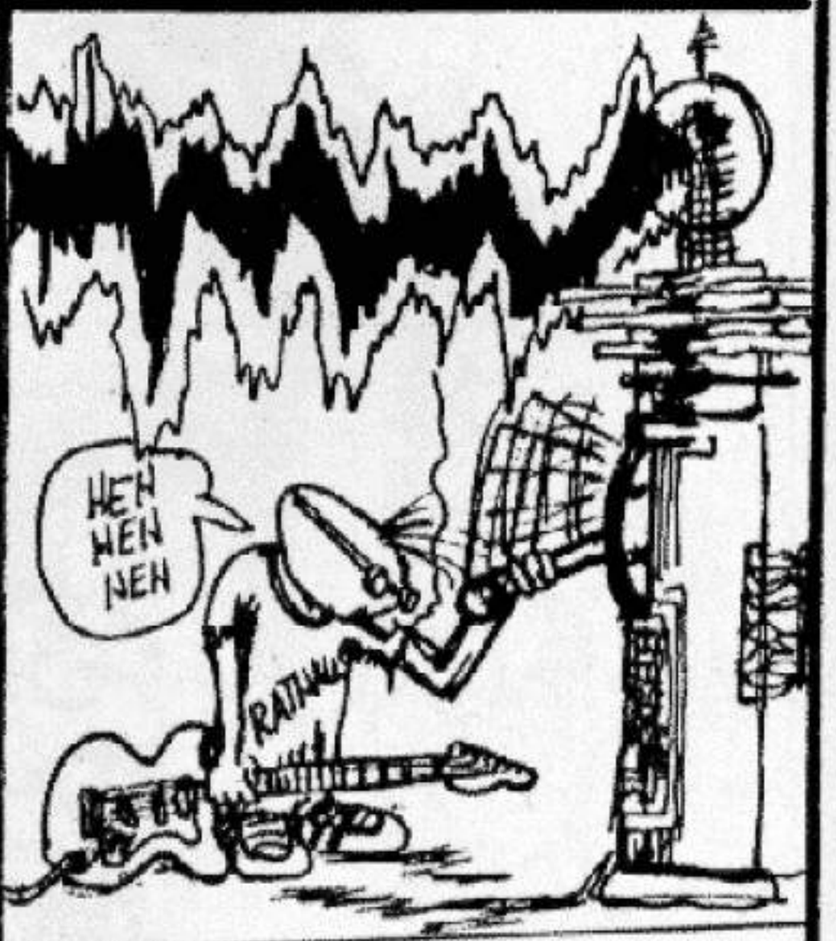


STEP II CONNECTING INTERFACE FOR EXTERNAL ANALOGUE INPUT USING ONLY A SOLDERING IRON AND HIGH RESISTANCE TELEPHONE CABLES.

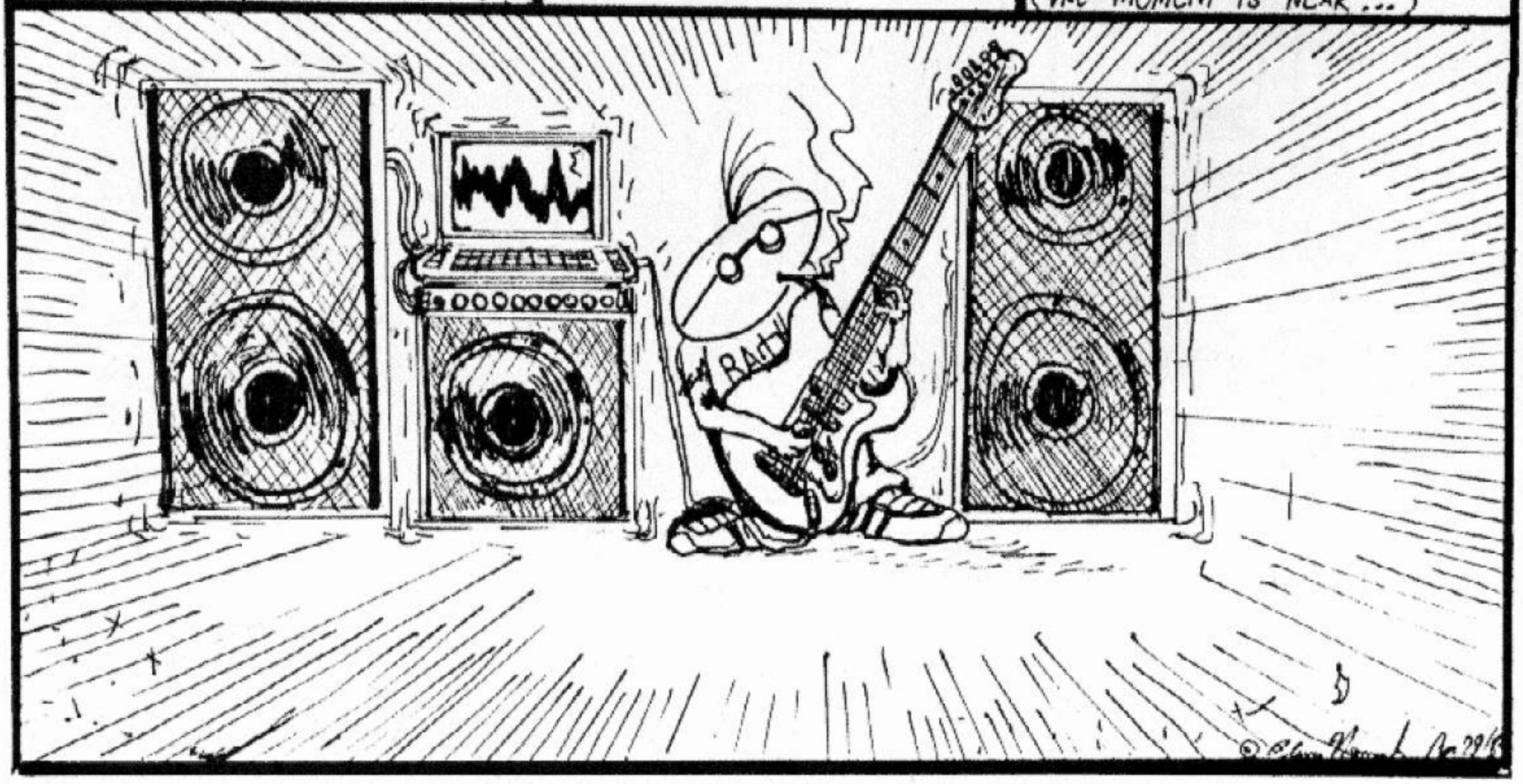


STEP III ROOTING THROUGH THE ARCHIVES, SEARCHING FOR AN ERGONOMIC USER-INTERFACE FOR THE SYSTEM...





STEP IV PULLING POWER LEVER TO "ON"; DRAWING MEGAWATTS FROM PICKERING FOR HIS OWN NEEDS. (THE MOMENT IS NEAR...)



The MANAGER Column

Don Bell
Milton, Ontario

Letters to the Manager

Many subscribers have written to me about their frustration with the manual and their desire to find more documentation for the program. Instead of writing everyone on an individual basis, I will use this column as my way of answering all your letters. First off, make sure you have the latest version of the '64 MANAGER, i.e. version 1.06b. Secondly, I am not aware of any other sources of documentation for the '64 MANAGER other than the manual and what you can glean from my columns. If I feel there is sufficient demand and I have the time and energy, I will attempt to put together a booklet of additional documentation.

Subjects covered in previous '64 MANAGER columns are as follows:

Vol 4 Issue 03	overview of '64 MANAGER technical details on the screen file
Vol 4 Issue 05	Gift (Mailing) List part 1 designing a screen in CREATE/REVISE option creating a file entering records
Vol 4 Issue 06	Gift(Mailing) List part 2 Potential Benefits of this Application Review of Screen Format Tricks for Entering Records Quickly Searches
Vol 5 Issue 01	Gift(Mailing) List part 2 Designing Reports Using Report Generate

Nelson Fung of Dallas, Texas wants to know how to modify the program to produce an 80 column printout. The Commodore '64 is a 40 column computer so you will never be able to get an 80 column display or report on the screen. However, you can get 80 column reports on a printer. In the Report Generate option, specify output to printer, line length 80, lines per page 66. Of course, your report must be designed as an 80 column report. When describing the LISTZONE the column number references must reflect the starting column positions of each print area in the LIST ZONE.

A few answers for Pete Musselman of Royerford, PA. No you cannot use two 1541 disk drives. The section on 'Accumulate' on page 14 of the manual has an example of an arithmetic file. Do not let it throw you, it's just an example. The 'accumulate' function can only be used if you have used the 'ARITHMETIC' option to create an arithmetic file. For example, in this issue there is a checkbook application. In order for the application to calculate a running balance, we had to do some arithmetic. This arithmetic was created in ARITHMETIC OPTION and stored in an arithmetic file. Now when we request 'accumulate' in the ENTER/EDIT option, the program will quickly read through all the records, select records satisfying the search criteria, perform the arithmetic on these records, and display the balance at the screen location specified in Arithmetic.

A.W. Lauer of Fredericksburg, VA wants to know how to begin a search at a particular record number. Unfortunately, the only direct way of doing this is by inserting a numeric field for the record number in your screen design. Then using the search string editor (F5), indicate records greater than a specified record number. Another way around this problem might be if there is a date field and records are usually entered in order of date. Sort on the date field, then

the search criteria will be greater than a specific date. If you have an alphabetical key you are sorting on, then you could specify records greater than 'N', for example.

Warren Knighton of Wassau, WI. had problems with the Rearrange a File function in the MANIPULATE files option. This is the function you will use if you want to change your file design. Unfortunately, I've also had several problems with this option as well. If the program stops before completion, before trying it again do the following. Load the Enter/Edit option and see how many of the records got transferred from the old file to the new file. If most of them are there, then you may be OK. The next problem is that they may appear to be there, but when you do a search for specific records they don't show up. To correct this problem use the Fix a File function in the MANIPULATE FILES option.

Richard Boisvert of Woonsocket, R.I. has been using the '64 MANAGER as a detective for the Woonsocket Police Department. Each record in his file has 6 fields in which to list stolen objects. When a stolen object is recovered he wants to search all 6 fields at once. Refer to the section of the manual on the 'hunt' option in the search criteria (p.27). The important points are (1) the use of complex search criteria (F5); (2) the use of 'H' to specify hunt for the object anywhere in the field; and (3) use of the logical operator 'OR to include many different searches. To locate a stolen object (e.g. Atari game) which could be at any location in any of the 6 fields, you would use the following complex search criteria (F5). H1'ATARI' OR H2'ATARI' OR H3'ATARI OR H4'ATARI' OR H5'ATARI' OR H6'ATARI'. You can also get more complex searches by looking for a number of objects at once. H1'ATARI OR H1'SHOTGUN' OR H2'ATARI' OR H2'SHOTGUN'. . . If the description of an object is ambiguous or you did not always describe it in the same way, you could again use the 'OR' statement to hunt for the object under different descriptions, e.g. H1'WALKMAN' OR H1'CASSETTE'.

An Introduction to ARITHMETIC

Don't be frightened away by the description of the ARITHMETIC option in the manual. I know at first it looks like advanced math, but once you grasp some basic concepts it's fairly simple.

The search criteria and the Arithmetic work together to give you very powerful accumulate and report possibilities. Using the Arithmetic editor in the ARITHMETIC option, you can describe what calculations you want to do on fields in your records. Upon exiting the option, a special file (AR.filename) is created where your arithmetic is stored. Whenever you are doing a search, accumulate or report, the search criteria select relevant records, then the Arithmetic file operates on those records calculating new amounts and displaying them at predefined places on the screen.

In the ARITHMETIC option you define bins or registers where you want to store numbers. You can perform arithmetic on numbers in the registers and you can display the contents of the registers at any location on any of your screens.

Why would you want to use ARITHMETIC? You might want to accumulate a total of the amount fields in our records. We might want to know subtotals as well as totals at break points in our reports. Sometimes it is desirable to transfer data from one screen to another. There are of course limitless reasons for wanting to use Arithmetic to create new information about your file.

Checkbook Application With ARITHMETIC

I have chosen to do a checkbook application as the Arithmetic logic is fairly simple.

You may have noticed the checkbook application in the back of the '64 MANAGER manual. It has a nice–looking colourful screen and includes most of the information you want to store about your cheques. Initially I was just going to use the application as is, go over how to copy the screens, reports and arithmetic. However, once I got working with it on a practical level (I always try to make an application work in the real world!), I found it had several limitations. The way it works, it could only handle 2 kinds of transactions – cheques and deposits. What about withdrawals, bank charges, loan and bill payments? Another problem is that the balance just shows up in the middle of the screen without any heading indicating that it is indeed the balance. Of course, like the other applications included on the disk, there is no indication of how to use the application.

This article is an attempt to overcome the deficiencies of the original checkbook application. This application should provide a workable tool for verifying your bank statement, balancing your checkbook, and generating numerous reports about your bank account.

Screen Design

I have redesigned the screen to enable the distinction between 5 different kinds of account transactions – bank charges, loan payments, cheques, deposits and withdrawals. In accordance with these modifications, I have also

redesigned the 'arithmetic' that calculates the balance. The balance is displayed in the screen heading next to the title "BALANCE" so you'll know what it is when it magically appears. I dropped the field "Signed By" as I do not have a joint account. If you have a joint account, you may want include it.

While on the subject of screen design, let me give you some reasons for my screen design. One of the big factors to consider is a design that facilitates minimal keystrokes. For example, not all records require entries in the fields for CHECK#, WRITTEN TO and DESCRIPTION. Therefore I put them at the bottom of the screen so the user would not be forced to cursor over these fields if they are not relevant.

Secondly, fields that need to be updated or revised should also be near the top of the screen. The OUTSTANDING field will have to be updated each time you get your bank statement (i.e. change the status of transactions from outstanding to not outstanding). Thus I chose to make OUTSTANDING the first field in the record.

A third factor in screen design is the visibility of the most important pieces of information – in our case the balance and the amount. These 2 pieces of information were place in very prominent positions in the top left corner of the screen.

Another factor in screen design is the correlation between the sequence of data entry on the screen and the sequence of information in the source document. In this case a compromise had to be made to accommodate the other factors. The sequence of entering data on the screen is almost the same as reading it from your checkbook with the exception of AMOUNT.

Finally, the screen should be visually pleasing to look at. Text and fields should be easily read, and spaced for clarity. Field prompts should not be ambiguous. The items on the screen should be well–balanced, reflecting the rules of good picture composition. Follow your creative instincts to brighten the screen up with color. There are too many dull computer screens out there already!

The Enter/Edit Screen

The following illustration shows the Enter/Edit form that was designed in the CREATE/REVISE option. Field lengths and field types (A = alphanumeric N = numeric) are shown. Note: there is no field under the title 'BALANCE'; this place (starting at line 3, column 31)is reserved as a display position for the account balance.

```
ROYAL CHECKING      ACCOUNT#      503-209-9
DON BELL            BALANCE       $

OUTSTANDING(Y/N)    DATE          AMOUNT
     [1A]         [↑  6N  ↑]    [$↑  7N  ↑]
                  TRANSACTION
                     [1A]
B = BANK CHARGE  C = CHEQUE  D = DEPOSIT
L = LOAN PAYMENT             W = WITHDRAWAL

                  EXPENSE TYPE
                    [↑2A↑]
AUTO              FOOD          MORTG./RENT
CASH              INSURANCE     REPAIRS
CLOTHING          MEDICAL       TAXES UTILITIES
CONTRIBUTIONS     MISC.         TELEPHONE

CHECK#                    WRITTEN TO
[↑  4N  ↑]                [↑          25A          ↑]

                  DESCRIPTION
[↑                   38A                          ↑]
```

Checkbook Arithmetic

Load the ARITHMETIC option from the main menu. You are now going to indicate that you want to have only one display position on the screen where the result of your arithmetic (the balance) will be displayed. You will also describe the exact screen location and length of the display position. In our case, we want to display the balance for our account next to the title 'BALANCE' in line 3 of our Enter/Edit screen. Complete the screen as below:

```
                    ARITHMETIC

NO. OF DISPLAY POSITION ON SCREEN 1?  1

   1. LINE? 3      COLUMN? 31      LENGTH? 9
```

Press back arrow (←)

We only use one register, R1, to store the current balance in our account. The object of our arithmetic is simply to calculate the current balance, move it into the register R1 where it is stored and then display it next to the title 'BALANCE' on the screen.

When calculating the balance in our account, we are dealing with several different kinds of transactions – deposits, checks, withdrawals, bank charges, etc. When looking at all transactions as a group, they all do one of two things – they either add to the balance or they subtract from it. The only transaction that adds to the balance is a deposit. All other transactions are subtracted from the balance.

Now you are in the EDIT MODE of Arithmetic. You will enter the arithmetic logic for calculating the balance. Complete the screen as below. Only deposits are added to the balance, all other transactions are subtracted. Therefore our logic is that if the transaction is a deposit (if field 4 = 'D') then add the amount in field 3(N3) to R1, ELSE (otherwise) subtract the amount in field 3 (N3) from R1. The ENDIF is required to complete the previous IF statement. Note: all lines beginning with ';' are documentary comment lines not required to execute the Arithmetic.

```
;CALCULATE BALANCE
;
;IF A DEPOSIT ADD IT TO R1
IF (F4 = 'D') THEN R1 + N3 TO R1
;
;SUBTRACT ALL OTHER TRANSACTIONS
  ELSE R1 – N3 TO R1
;
ENDIF
;
;MOVE THE BALANCE IN R1 TO DISPLAY
;POSITION 1 WITH 2 DECIMAL PLACES
R1 TO 2D1



EDIT MODE
```

Press back arrow

```
CHECKING STRUCTURE
ARE YOU SURE(Y/N)?
Enter 'Y' <RETURN>
STORING MATH
```

Making Field Entries in the Enter/Edit Option

On the Enter/Edit menu at the bottom of the screen 'E' is for entering/adding a new record. If you want to change a record you first have to get the record ('G') then change it ('C').

Field 1 (OUSTANDING(Y/N) requires a 'Y' or 'N' entry. Field 2 requires a 6 digit number for the date in the form YYMMDD, e.g. 840404 for April 4, 1984. One letter codes (e.g. D = DEPOSIT) are entered in the Transaction field. Two–letter codes are entered in the Expense Type field (e.g. AU = AUTO). Remember to use F3 to duplicate an entry in the previous record, or Shift 'E' to duplicate an entire record already on the screen. Often it is faster to duplicate a similar record you have already created and then modify it, then it is to type in a whole new record.

Setting Up the File Using A Bank Statement

Record 1 must have starting balance for your bank account in the 'amount' field. Begin with your last bank statement balance. Enter the 'Balance Forward' amount from the previous statement as the 'amount' in record #1 and record the transaction as 'D' for deposit. Then enter all the subsequent transactions on the bank statement. As the bank knows about all these transactions they will all be entered as NOT outstanding (i.e. field 1 will always be 'N') . Don't forget bank charges, loan payments, withdrawals, etc. NSF cheques are treated just as they are on your bank statement i.e. they are entered twice – first as cheques and then as deposits since they are returned to your account. (We're going to first check are arithmetic against theirs.) When you've made all the entries exactly like theirs, then try an accumulate. The final balance should be the same as on the bank statement. If there is a problem, check the records and the arithmetic. It's unlikely their computer made an error, but it is possible. If worse comes to worse, check the statement with your calculator.

Now using your checkbook, enter all outstanding transactions that have not appeared on any of your bank statements. Make sure field 1 is 'Y' for all these transactions. You will now have entered in your file all relevant transactions since your previous bank statement.

If you wish to get the current balance in your bank account, do an 'accumulate' with no search criteria. If you wish to test the bank statement balance again now that you have entered all the transaction records to date, do an 'accumulate' specifying only transactions that are not outstanding (i.e. F1 = 'N' is the search criteria).

Updating the File Each Month

1. When you get your next bank statement is a good time to update the file. Start by entering all new transactions from your checkbook('Y' for OUTSTANDING). Make sure you include all checks, deposits and withdrawals.

2. Now enter into your MANAGER file any transactions that

appear on your bank statement that are not in your checkbook, e.g. bank charges and loan payments. They are entered as NOT OUTSTANDING in field 1. Your MANAGER file should now contain all transactions to date.

3. Now you are going to search for all outstanding transactions in your MANAGER file that also appear on your bank statement. If an outstanding transaction in your MANAGER file also appears on your bank statement, then you will have to change the status of the record to NOT OUTSTANDING. Note: during this process the balance shown on the screen will not be correct. Do not worry about it.

While in the Enter/Edit option:
Place cursor in field 1
Press S (for search)
Enter 'Y' in field 1
Press F3 (indicates a position dependent search)
Press back arrow. (to execute the search)

You will now be presented with the first outstanding transaction. If it appears on your bank statement, checkmark it on the statement and do the following:
Press 'C' for change
Change 'Y' to 'N' for OUTSTANDING.
Press back arrow to store the revised record
Press space bar to find the next outstanding transaction

If a transaction does not appear on your bank statement, it is still outstanding. Do not change anything. Press space bar to view the next outstanding transaction. . .and so on.

Keep repeating this updating process until all the transactions in your MANAGERfile that are also on your bank statement have an 'N' instead of a 'Y' in field 1.

Verifying the Bank Statement

Now you can verify the bank statement balance using your MANAGER file.

While in the Enter/Edit option do an 'Accumulate' specifying only transactions that are not outstanding (i.e. search criteria is F1 = 'N')

Type 'A' for accumulate
Press F5 to access the search string editor.
Type F1 = 'N'
Press back arrow

Getting the Current Balance in Your Account

If you wish to get the current balance in your bank account, do an 'accumulate' with no search criteria. This will accumulate amounts in all records in the file.

Note: 'Accumulate' rapidly executes the search criteria and the arithmetic without forcing you to examine each record in detail. It is very handy for accumulating a total amount.

Useful Searches

One of the big advantages of having your checkbook electronically filed is that you can rapidly search for and find individual transactions. For example, you might want to look for all checks written for your auto maintenance. In the Enter/Edit mode press 'S' for search, enter 'au' as expense type, press F3 for a position independent search, press back arrow. Press space bar to get the next record, and so on. Similarly, you could search for any other expense type. You will also notice that the balance displayed will be the accumulated amounts of the search as a negative amount. If you only want the total amount spent for a particular expense type, specify 'accumulate', then enter your search criteria, then press back arrow.

Using the search string editor you can specify more complex searches. For example all checks written to someone after a certain date. Press 'S', F5 (for search string editor), enter search criteria, then press back arrow. For example, to search for checks written to 'VISA' after the last day of February, the search criteria would be:

F7 = 'VISA' and N2>840229.

Generating Checkbook Reports

I have designed two report formats, one for the screen and one for the printer. Using these basic report formats you will be able to generate numerous reports simply by changing the search criteria and header description of the report. Reports that you might want to generate could include: a listing of all outstanding transactions; a summary of one or more expense types for a certain time period; a list of all checks written to one company.

Refer to the previous MANAGER Column (Vol. 5 issue 01) for more details on using the Report Generate option.

Reporting to Screen

I designed my 40 column screen report on graph paper to look like this:

```
                          Column#
              1    6       13        22       32  35
HEADER ZONE
line #1       ROYAL CHECKING ACCOUNT# 504-209-9
      2
LIST ZONE
line #1       REC# :23     BALANCE :$  250     OS:Y    (Print Areas 1-4)
     #2       DATE:849508  AMOUNT :$  75               (Print Areas 5-8)
     #3       TRAN:CH      TYPE     :  AU              (Print Areas 9-12)
     #4       TO    :BENDER AUTO PARTS                 (Print Areas 13-14)
     #5       CAR REPAIR                               (Print Areas 15)
     #6
```

Line 5 of the LIST ZONE has no title. It is the description of the transaction in field 8. The List Zone has a total of 6 lines and 17 print areas. Here is a summary of relevant entries for the LIST ZONE. Use the defaults for the remaining entries.

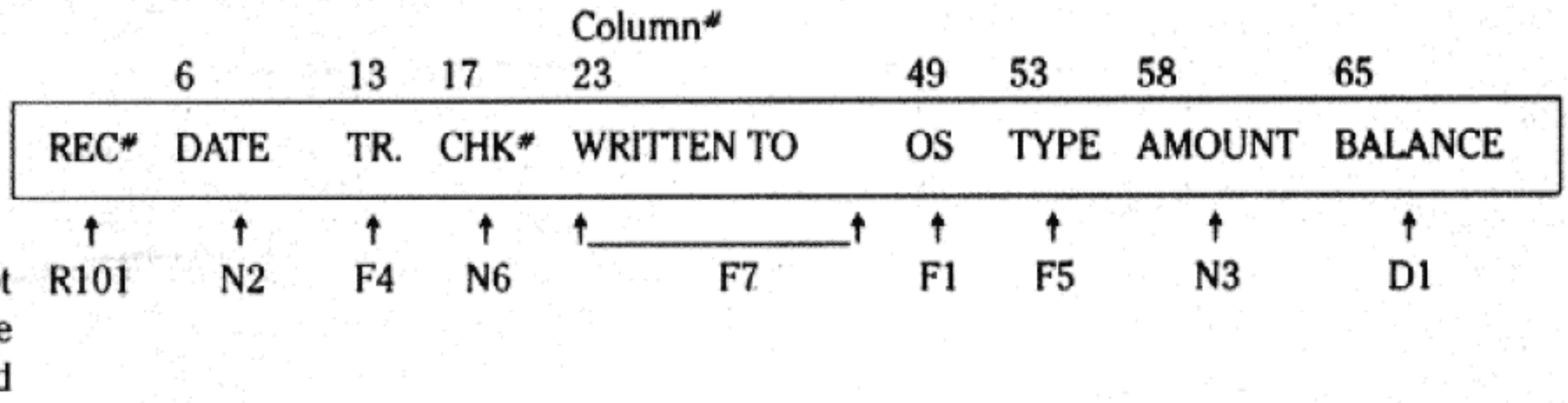
Print Area#	Data Type	Sub-script	Text/Title	Area Length	Line#	Col#
1	T		REC#	5	1	1
2	R	101		3	1	6
3	T		BALANCE:$	9	1	13
4	D	1		9	1	22
5	T		OS:	3	1	32
6	F	1		1	1	35
7	T		DATE:	5	2	1
8	F	2		6	2	6
9	T		AMOUNT :$	9	2	13
10	F	3		7	2	22
11	T	4	TRAN :	5	3	1
12	F	4		1	3	6
13	T		TYPE :	9	3	13
14	F	5		2	3	22
15	T		TO :	5	4	1
16	F	7		25	4	6
17	F	8		38	5	1

Reporting to the Printer

A sample heading in your header zone might look like this:

```
ROYAL BANK CHECKING ACCT# 504-209-9
     OUTSTANDING TRANSACTIONS
```

Here is a one line LIST ZONE format that includes all fields except description. (TR. = Transaction OS = OUTSTANDING?)



In order to produce the printer report format above, make the following screen entries in the REPORT GENERATE LIST ZONE.

Print Area#	Data Type	Sub-script	Text/Title	Area Length	Line#	Col#	Center	# of Dec.
1	R	101	REC#	4	1	1	N	0
2	F	2	DATE	6	1	6	N	0
3	F	4	TR.	1	1	13	N	0
4	F	6	CHK#	4	1	17	Y	0
5	F	7	WRITTEN TO	25	1	23	N	0
6	F	1	OS	1	1	49	N	0
7	F	5	TYPE	2	1	53	Y	0
8	F	3	AMOUNT	7	1	58	Y	2
9	D	1	BALANCE	9	1	65	Y	2

More Complex Possibilities

As you enter more and more transactions into your MANAGER checkbook file, it will take longer and longer to accumulate the balance. A way around this problem is to create a new status for the OUSTANDINGFIELD for transactions that are on your current bank statement. Instead of being labeled as 'Y' or 'N' in the OUTSTANDING field, you could temporarily label them as 'S', indicating they are on your current bank statement. In record 1 enter 'S' for OUTSTANDING, the Balance Forward from the previous statement as the 'Amount' and 'D' for a deposit transaction. Then change the status of all transactions in your file that are also on your statement from 'Y' to 'S'. Now you can do an 'accumulate' specifying 'S' in field 1 as your search criteria. The final balance should be the same as your bank statement. If your balance is off from the statement balance use the search function to track down transactions by check#, amount or outstanding status.

Once you've got your bank statement verified, you can change records with 'S' in the OUTSTANDING field to 'N'. Globally update all 'S' status records to 'N' in the outstanding field. (Press Shift 'C' – Prompt:Change Field Number – Enter '1' – Enter 'N' in field 1 – Press back arrow – Prompt: Change Field Number – Press back arrow – Prompt: Accumulate – Enter 'S' in field 1 as the search criteria – Press F3 – Press back arrow – see manual p. 15).

If your last bank statement verified correctly, you can now get your current balance by entering the final balance on your bank statement into the amount field in record 1 and the status as 'Y' for outstanding. Then do an accumulate specifying field 1 = 'Y'.

This application could be revised to keep track of a credit card account instead of a bank balance.

DON'T PHONE – WRITE!

If you have questions regarding this application or you would like to submit your own "terrific" application, please write me a legible, coherent letter. If you submit an application, send it on disk or at least send screen dumps of the ENTER/EDIT screen, a hand–drawn report chart and any math and sample data. I will attempt to answer letters in this column. Write to: Don Bell, c/o The Transactor, 500 Steeles Ave., Milton, Ontario, Canada, L9T 3P7.

MAILPRO 64: A Review

Chris Zamara
Downsview, Ontario

Mailpro is a file management program for use on the C64 and 1541 disk drive. Written by Steve Punter, author of the WordPro series, Mailpro is ideally suited for keeping mailing lists (as its name suggests), telephone numbers, or similar lists for home or small business. Mailpro allows such lists to be created, updated, and printed in any format, and in any order. Data for a list may be entered using Wordpro, and Mailpro will read the Wordpro file: a time saving feature for long lists.

First, the documentation. The manual is packaged in an attractive and sturdy little vinyl binder like the other PRO-LINE software products. My main critisism of the manual is its lack of a reference section containing a summary of all single key control functions. Using the package for the first few times may be a bit frustrating, since you will have to keep skimming through the text to find out how to access special functions while outside the main menu. The manual is, however, reasonably well organized, with separate main sections dedicated to creating, updating, and printing a file. The text itself is generally understandable, but muddled in some sections. I would rate the manual a 6 out of 10.

As for the program itself, it should first be realized what Mailpro's intended use is. This package does not have the features and flexibility of a full-blown database management system, but it is not designed as such. The documentation claims Mailpro to be a "simple to use but sophisticated mailing list program approaching a full fledged data base in capability". It goes on to suggest that name and address lists are a "natural" application of Mailpro. For this application, Mailpro lives up to its claims, and certainly does everything you could ask from a mailing list program. Mailpro will not, however, generate complex reports and process data contained within fields, nor will it do any sort of automatic updates on any records. Mailpro has all of the capabilities one can expect from a package at its price– around $70.

I told you what it doesn't do. Now what does it do? The main menu gives 12 options, including creating a file, entering records, recalling specific records, and setting up a print format.

The file create editor allows a record to be set up in the format that data will be entered (this is not necessarily the output format of the record). This editor is very easy to use, and works like Wordpro, allowing full cursor control. Labels up to 12 characters long can be entered, and the position, length, and type of fields are defined. The label length limitation is not so bad, since these labels are only used as a reminder of which field is which when entering the data into each record.

"Add New Record" mode allows new records to be entered into the file, using a similar free-form cursor oriented editor to fill in the fields. A control key submits a record to the file once you are satisfied with its contents.

In "Recall a Record" mode, a specific record can be searched for using any "sort" field as a key. The next record or previous record in alphabetic sequence can be viewed by pressing control keys. Records can also be deleted and

updated in this mode. Deleting or updating a record takes a while, but the slowness probably has more to do with the speed of the 1541 disk drive than with Mailpro.

The feature I was most impressed with is the "Setup Editor" mode. The setup editor allows you to custom tailor the output of the records in a file, and it gives total flexibility. A setup file is created using a very Wordpro-like editor to position labels and fields anywhere on the print page. The screen scrolls left and right with the cursor, allowing text up to 160 characters wide on a page. All record fields must be defined as left or right justified, or compressed. Once a field is defined, scrolling over the field with the cursor will reveal the field type in a status line at the top of the screen. Very nifty. A heading of any number of lines to top each page can be provided, along with a page number, if desired. Record numbers can also be printed anywhere on a page. Additional information about the output records, such as width, height, how many across on a page, etc. can also be supplied from this mode. Up to ten printer setup files may be stored along with a file. Overall, the output formatting is the most flexible and powerful feature in Mailpro.

When actually printing records (by selecting "Output to Printer" from the main menu), boolean decisions can be made to selectively print records (inclusive, exclusive, inside a range, or outside a range). For example, you could print all records with the second field beginning with "z", or print all records in sequence from "a" to "z". There are many nice little features incorporated as well, such as giving a printout of input record formats or setup files, and the use of default information in entering records. Lots of features, however, also means remembering lots of control sequences, or frantically scanning through the manual to find out how to do something. I suppose though, like any complete software package, it just takes a little more time to become totally familiar.

A Few Miscellaneous Gripes:

1) After certain operations, such as adding records, the drive error light remains blinking. The manual says this is normal, but it drives me crazy wondering if it is a serious error and I have lost all of the data I have entered (which happened to me once).

2) When creating a new file, an old file of the same name will be replaced without any warning. This could be disastrous!

3) As mentioned previously, the multitude of control features in different modes could be confusing, at least at first, and there is no summary of what does what in which mode.

4) Disk operations can sometimes leave you waiting a long time. Mailpro adds records to the file in a batch after you have entered all records (up to 127 at a time), not one at a time as they are entered. The manual states that writing 127 records to a complex file could take up to 2 hours.

If some of the above gripes sound like nit-picking to you, then that's good. They are the things that bothered me most about using Mailpro.

Some Exceptionally Good Points Worth Mentioning:

1) The "Setup Editor" for setting up the output format is fantastic. It's very easy to use, and allows total flexibility in formatting your output. The only limitation is a maximum length of 25 lines for each record, but that shouldn't be a problem in most cases. The fact that you can save up to ten setup files with each file is also a nice feature.

2) Mailpro is designed to be compatible in many ways with Wordpro, and data from a Wordpro file can be used to fill mailpro records. Conversely, the output from Mailpro can be sent to disk instead of printer in a format such that Wordpro can use the data as variables. Thus, if Mailpro holds your mailing list, you can print form letters to selective members of the list with Wordpro. The disk output feature could also be used to transfer data from one Mailpro file to another.

3) The "Index" function gives a list of all Mailpro files on disk, and nothing else, so you can use a disk with all kinds of stuff on it and only see what you need to from Mailpro.

Overall, Mailpro is good for keeping track of any household or small business lists you might have, and great at generating fancy selective printouts of the lists. It is fairly easy to use, and if you are familiar with Wordpro, it's use is quite natural. For the price of around $70.00, Mailpro is a good buy, and may get you to finally do all of those things you wanted to with your computer, like store your recipes, record collection, telephone numbers. . .

PERSPECTIVE:
To GET Or Not To GET. . .
How Useful A Trick?

Elizabeth Deal
Malvern, PA

INPUT or GET? Each is useful in its own way. Each needs to be used intelligently, matching the best features of the command to the task at hand. Much was written about the horrors of dropping out of INPUT on the PET when you strike RETURN. While this feature makes INPUT unsuitable for many serious applications, it is one of the nicest features in debugging or in a quick run of a utility program: you can get out at any time. Consider the "improved" INPUT in the VIC or the C64. Unless you carefully code (see article v4 #6 p36, for instance) all possibilities (null entry, escape sequence) even for the most trivial, tiny, routine, you're stuck in the input loop. Forever.

Press RESTORE/STOP you say? Sure. Now the screen is blinding blue so you type all the nasty POKEs to turn the blue off. Utilities have disconnected, so you type all sorts of code to hook them up again. Your carefully defined characters are gone, so you connect them back. The bit-mapped display setup is gone, so you poke some more. The sprites you had on the screen are gone, you gotta bring them back. The interrupt vector you changed is back to normal, so you change that. An alternate BASIC language you connected is gone, so you hook that up. . . some improvements we could do without.

Unbreakable input routines are vital in many instances, of course. Many good ones have been written and are in circulation. They are needed when the features of INPUT do not match what you need done. Most INPUT simulators use GET in some fashion, with a graphic character used for the missing cursor. A well-designed GET routine is capable of tracking your typing in all directions. Other routines may need to prevent the user from typing, say, up or clear-screen. It all depends on the application. But the key feature of good routines is that they work.

On the other hand, tricks such as the old POKE 167 (originally introduced, I believe, in the Osborne's PET guide, subsequently reported in COMPUTE and most recently in The Transactor v4 #6 p37) pose hazards of which you may not be aware. Let's look at it a bit more closely.

POKE 167 was meant to put the cursor on the screen during a GET-type of input. A little fiddling with this trick shows that it only works in forward typing. You can't delete/correct your input and you can't go up or down to, perhaps, input from a different line on the screen. It leaves REVERSE FOOTPRINTS all over the screen at, seemingly, unpredictable moments. Useless.

The reason behind the failure of POKE 167 is that there is much more to the functioning of a cursor than just switching it on or off. A careful reading of the Butterfield memory maps, or even a superficial reading of the IRQ service routine in the computer reveals that enabling a cursor must be done IN THE RIGHT PLACE and AT THE RIGHT TIME. Poke 167 ignores that second, vital, condition.

You may wish to look up an old issue of COMPUTE in which Timothy Striker addressed this problem and introduced a wonderful routine which puts the cursor on the screen at the right time and takes it off at the right time. It's well worth typing in. There is one typo there, but you should be able to fix it after you understand the article.

Last, but not least, don't forget that the old screen and keyboard inputs can do wonders. The point is to use either one as a file, as in:

```
OPEN1,0 or OPEN1,3: INPUT#1,I$: CLOSE1
```

It fits numerous applications quite well and is machine-independent. It is up to you to decide if you can get away with that or if you must go the, more complicated, but foolproof, GET route.

All About Commodore BASIC Abbreviations

Louis F. Sander
Pittsburgh, PA

Many Commodore owners know that some of their BASIC keywords can be abbreviated to ease the work of program entry. Those who know about the abbreviations often don't use them extensively, because it's hard to find them listed together in one place. Furthermore, they aren't all constructed in the same way, and it's hard to remember the difference, for instance, between the abbreviations for RESTORE and RETURN. This article will end the confusion. It provides alphabetized tables of all known abbreviations, including versions for all ROMs and character sets. For folks just learning about Commodore abbreviations, it contains a tutorial on their use.

Except for the familiar question mark used for PRINT, all Commodore abbreviations consist of one or two unshifted letters from the keyword, plus one SHIFTed one. Table 1 shows the screen display for keywords and their abbreviations when the graphics character set is enabled. Tables 2 and 3 show the same display when the 'lower case' character set is enabled for original PETs and for newer machines. The keystrokes used to produce each table are exactly the same; as long as the proper shifted and unshifted keys are pressed, it doesn't matter what comes up on the screen. We have printed several tables only to make abbreviating easier for you — use whichever one you find most convenient. We've also listed the BASIC 4.0 disk commands separately in Table 4, to keep things simple for those who do not use them.

Now look closely at the tables and notice that most abbreviations consist of keyword's first letter, unshifted, plus its second letter, shifted. Others have two unshifted letters before the shifted one; this happens where the first two letters of several keywords are identical, as in STEP and STOP. Some statements in the tables, (e.g. COS), are followed by a double dash. We know of no abbreviations for these, in spite of having searched diligently for them. (Let us know if make any discoveries) and look carefully at the abbreviations for SPC and TAB. They are actually abbreviations for 'SP(' and 'TAB('. If you put tA(20) into a program line, it will be interpreted as tab((20), and you'll get a ?SYNTAX ERROR when the line is executed.

It's not widely known that many abbreviations also have longer forms. For these, you can type as much of the keyword unshifted as you'd like, then shift the next letter and stop. The computer will recognize your intent and fill in the blanks accordingly. You can verify this for yourself by entering 'r–i–shifted–g' instead of the more familiar 'r–shifted–i', and seeing that both forms produce a 'right$' when listed. We haven't tested this on every keyword, but we haven't ever found it to fail.

Now for something about using the abbreviations. They work equally well in direct or program mode, as most of us know from our experience with '?' as used for PRINT. Using them in program lines does NOT save any memory, in spite of what you may have read elsewhere. It only saves keystrokes and space on the line originally entered. Enter a short program with, and then without, abbreviations, PRINTing FRE(0) each time to prove it to yourself.

When your machine LISTs a program line that was entered with abbreviations, it spells the keywords out in full. This

principle should also be familiar to anyone who has used a question mark as shorthand for PRINT. What may not be familiar is best illustrated by an example. Enter one 60–80 character program line containing many abbreviations, such as

```
10 a=5:?a:?a:?a:?a:?a:?a:?a:?a:?a: . . .
```

Be sure the line you enter fills most of two screen lines (one for 8032's, four for VIC's). Now LIST your program. Surprise! The long line now fills far more than 80 spaces on the screen. RUN your program and observe that it executes perfectly, even though it seems to exceed the 80-character limit on line length. You can use this idiosyncracy to your advantage when trying to pack a lot of statements into one line. If what you want doesn't fit, just abbreviate some of the statements, so that the abbreviated line fits into 80 spaces or fewer. When you LIST the line, it will expand enough to spell out all the statements in full, but it will RUN perfectly. Do not attempt to edit any of these long lines after they have been entered. The screen editor will enforce the 80–character limit as soon as you use it, and your line will be truncated to 80 spaces on the screen. If you fail to allow for this, you'll find yourself puzzling over vanished parts of your cleverly–squeezed–in program lines.

That's about all there is to know about keyword abbreviations. Knowing about them has made my programming life a little bit easier, and my fingers a little less fatigued. I have fastened a copy of Table 3 to the front panel of my PET, where it serves as an easy reference to the abbreviations I haven't committed to memory. If you do the same, you'll bask in the benefits of using abbreviations.

Editor's Note

The abbreviations phenomena is actually the result of a bug! A quite harmless bug, but a bug just the same. Each keyword is held in ROM in a 'table'. The table is simply each keyword spelled out, one after the other. Naturally, the interpreter must know where one starts and the next begins, so the last letter of each keyword is OR'd with 128 (ie. Bit 7 set).

When you enter a command in its long form, the interpreter begins comparing letters to those in the table. The interpreter continues comparing until a mismatch is found. Assuming it was spelled correctly, when the interpreter gets to comparing the last letter, there will be a difference of exactly 128. A difference of 128 (Bit 7) signals the interpreter that a complete match has been made, so go on to the next step.

If you entered the same command in its abbreviated form, the character you type with the shift key will have a value exactly 128 greater than the same character 'un-shifted'. The interpreter detects the 128 difference and 'thinks' it has come to the end, so go on to the next step.

Inotherwords, it doesn't matter which character is responsible for the 128 difference when the compare is performed. See Mike Todd's article, "How BASIC Works" for more details on this anomaly.

The general rule of thumb, though, is the first letter followed by the shifted second letter, unless this matches something else first, then it's the shifted third letter. Two character commands like IF and FN have no abbreviation.

Recall I said this 'feature' is actually a bug. Really, the editor should force you to type the entire keyword (except ? for PRINT). Try entering a keyword with the LAST character shifted. Try for example:

```
10 nexT
```

Since the last character of 'nexT' will have the same value as the corresponding character in the table, no match will be found. The interpreter enters 'nex' onto line 10 as though it were the beginning of a variable. The shifted 'T' is lost because the editor doesn't allow shifted letters outside quotes (except on REMark lines, another anomaly).

Once you know how the keyword table is constructed, all kinds of neat (but useless) tricks can be played. In the keyword table, 'NEXT' is immediately followed by 'DATA'. Try entering:

```
10 nexTdaT
```

The interpreter finds the 128 difference on comparing the second 'T'. The string is replaced by the corresponding token, and LIST will show:

```
10 next
```

Try adding:

```
20 returN
30 gosuB
```

Now LIST and see if you can tell what the interpreter has done.

Table 1

ABS	A|	DIM	D┐	INPUT#	I/	ON	--	RIGHT$	R┐	TAB(	T♠
AND	A/		END	E/	INT	--	OPEN	O┐	RND	R/	TAN	--
ASC	A♥		EXP	E♠	LEFT$	LE▬	OR	--	RUN	R↗	THEN	T|
ATN	A|	FN	--	LEN	--	PEEK	P─	SAVE	S♠	TO	--
CHR$	C|	FOR	F┌	LET	L─	POKE	P┌	SGN	S|	USR	U♥
CLOSE	CL┌		FRE	F▬	LIST	L┐	POS	--	SIN	S┐	VAL	V♠
CLR	CL		GET	G─	LOAD	L┌	PRINT	?	SPC(	S┐	VERIFY	V─
CMD	C\		GET#	--	LOG	--	PRINT#	P▬	SQR	S♥	WAIT	W♠
CONT	C┌		GOSUB	GO♥	MID$	M┐	READ	R─	STEP	ST─		
COS	--		GOTO	G┌	NEW	--	REM	--	STOP	S|		
DATA	D♠		IF	--	NEXT	N─	RESTORE	RE♥	STR$	ST▬		
DEF	D─		INPUT	--	NOT	N┌	RETURN	RE|	SYS	S|		

Table 2

ABS	Ab	DIM	Di	INPUT#	In	ON	--	RIGHT$	Ri	TAB(	Ta
AND	An	END	En	INT	--	OPEN	Op	RND	Rn	TAN	--
ASC	As	EXP	Ex	LEFT$	LEf	OR	--	RUN	Ru	THEN	Th
ATN	At	FN	--	LEN	--	PEEK	Pe	SAVE	Sa	TO	--
CHR$	Ch	FOR	Fo	LET	Le	POKE	Po	SGN	Sg	USR	Us
CLOSE	CLo	FRE	Fr	LIST	Li	POS	--	SIN	Si	VAL	Va
CLR	Cl	GET	Ge	LOAD	Lo	PRINT	?	SPC(	Sp	VERIFY	Ve
CMD	Cm	GET#	--	LOG	--	PRINT#	Pr	SQR	Sq	WAIT	Wa
CONT	Co	GOSUB	GOs	MID$	Mi	READ	Re	STEP	STe		
COS	--	GOTO	Go	NEW	--	REM	--	STOP	St		
DATA	Da	IF	--	NEXT	Ne	RESTORE	REs	STR$	STr		
DEF	De	INPUT	--	NOT	No	RETURN	REt	SYS	Sy		

Table 3

abs	aB	dim	dI	input#	iN	on	--	right$	rI	tab(	tA
and	aN	end	eN	int	--	open	oP	rnd	rN	tan	--
asc	aS	exp	eX	left$	leF	or	--	run	rU	then	tH
atn	aT	fn	--	len	--	peek	pE	save	sA	to	--
chr$	cH	for	fO	let	lE	poke	pO	sgn	sG	usr	uS
close	clO	fre	fR	list	lI	pos	--	sin	sI	val	vA
clr	cL	get	gE	load	lO	print	?	spc(	sP	verify	vE
cmd	cM	get#	--	log	--	print#	pR	sqr	sQ	wait	wA
cont	cO	gosub	goS	mid$	mI	read	rE	step	stE		
cos	--	goto	gO	new	--	rem	--	stop	sT		
data	dA	if	--	next	nE	restore	reS	str$	stR		
def	dE	input	--	not	nO	return	reT	sys	sY		

Table 4

APPEND	A┐	COPY	CO┐	DSAVE	D♥	append	aP	copy	coP	dsave	dS
BACKUP	B♠	DCLOSE	D─	HEADER	H─	backup	bA	dclose	dC	header	hE
CATALOG	C♠	DIRECTORY	DI▬	RECORD	RE─	catalog	cA	directory	diR	record	reC
COLLECT	COL	DLOAD	DL	RENAME	RE/	collect	col	dload	dL	rename	reN
CONCAT	--	DOPEN	D┌	SCRATCH	S─	concat	--	dopen	dO	scratch	sC

©1982 LOUIS F. SANDER

How BASIC Works

Mike Todd
Kent, England

Mike Todd is a member of the Independent Commodore Products User Group of England. This article originally appeared in the club newsletter some four years ago. Essentially not much has changed since then. Commodore has added here and taken away there, but the fundamental operation of all their machines (excluding the B and the new 264) is still the same. You might consider having your memory map on hand as you read through this article. As you become more familiar with your machine you'll find that writing new and more complex programs becomes easier and easier. M.Ed.

It was difficult to decide at what level to aim this article, and I have attempted to keep it fairly simple (and consequently omit a lot of detail – for which I apologise in advance). The only requirement is that the reader understand the principles of machine code programming and have some knowledge of simple terminology such as RAM, byte, stack, hex notation and so on.

The Inside Story

The first consideration must be to sort out the organisation of the massive amount of software in ROM. There are three main divisions:

1) The Operating System – is the section of ROM dealing with cassette, keyboard, screen and IEEE input or output. It is written specifically to match the hardware.

2) The BASIC Interpreter – written by Microsoft, this is the program which allows you to write lines of program, edit them, and execute them. It was originally written in general terms for use by different computers and is customized by Commodore for use on their machines. In fact, most of the interpreter is identical to that in other machines such as the APPLE.

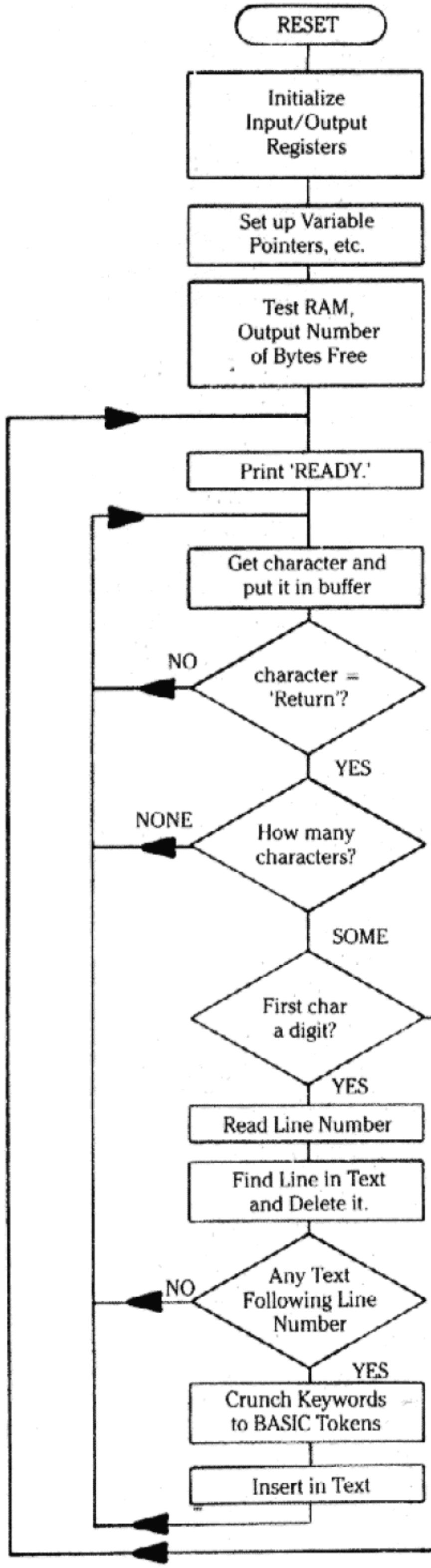
3) The Machine Language Monitor – although not really part of the normal operation of the PET, this program provides facilities for manipulating machine code programs using hex codes and for saving and loading machine language programs from disk or tape. On BASIC 1 PETs, this is replaced by diagnostic routines which are of very limited use to most owners. Therefore, a MLM program had to be loaded into RAM to obtain this facility. Likewise on the VIC and 64, however cartridges are available too that contain this program.

At the heart of the machines is the microprocessor which at switch–on starts to execute the ROM software at an address placed at the end of ROM ($FFFC/D). This routine initializes all registers, input/output chips, BASIC pointers and vectors (a vector is an address held in RAM for future use and which indicates the entry point of a routine held somewhere in memory). It clears the screen and checks RAM by writing a number into every location and confirming that it is read back correctly. It does this twice for every location and as soon as it detects an error it assumes that it has run out of RAM and sets the end of RAM pointer accordingly. The greeting message is printed using this information to indicate the number of bytes free. Then the READY message is printed, the cursor is flashed and a holding loop is entered waiting for you to type something.

Interrupts

While waiting, the microprocessor is far from idle. Every 1/60th of a second, a pulse is generated and fed to the Interrupt Request pin (IRQ). When the pulse is received, the 6502 (6510 in the 64) stops whatever it is doing, saves the status register and return address and starts executing code at the address given in $FFFE/F. This routine saves the registers, does some housekeeping and then uses the IRQ vector to jump to the main interrupt routine. Since this vector is in RAM, the user can alter it to point to his own interrupt routine – the only restriction being that it should end with the same code that the regular interrupt routine uses. Here the registers are restored to their previous conditions, followed by a Return from Interrupt instruction (RTI).

The first task of the main interrupt routine is to execute a subroutine that updates the TI clock and sets a flag if the STOP key is pressed. Bypassing this subroutine (by changing the IRQ vector to enter the interrupt routine just beyond this JSR) is one of the ways in which the stop key can be disabled. However, this also stops the TI clock.



The interrupt routine then looks after the cursor on the screen: if a cursor is required (flash cursor flag = 0, see your memory map) a countdown is maintained and on every 20th interrupt, the character under the cursor is inverted, making the cursor flash 3 times a second. The cassette buttons are then checked and the motors turned on or off as required. This is necessary since the motors are totally under software control and need to be switched on and off to allow wind or unwind to function correctly.

Finally, and probably the most important interrupt function, the keyboard is scanned and if a key is pressed, the ASCII value for the key is found in a table and placed in the keyboard buffer. This buffer is 10 bytes long (which can be altered in most models), and a count is kept of the number of characters in the buffer. If this count goes over 9, the buffer is full and the interrupt routine stops accepting characters. By holding characters in this way, the user can type while a program is executing some task. Then when the program comes to an INPUT or GET statement (or direct mode), the keyboard buffer empties its characters onto the screen ready for use.

This process continues nearly all the time. However, an instruction is available (SEI) to turn this interrupt off and this is done now and again whenever time is critical. In addition, cassette input and output uses a different interrupt facility and keyboard servicing is bypassed altogether.When the cassette has finished with the interrupt, it restores the vector. In PET/CBMs the vector is restored to the main interrupt routine, which means if the user has altered the IRQ vector it will require "re-altering". VIC and 64s restore the vector to its previous location even if it was altered by the user.

Ready and Waiting

Armed with this information we can now rejoin the main holding loop just after READY has been printed out. The first thing that has to be done is to accept characters from the keyboard (more precisely, from the keyboard buffer) and print them to the screen along with a flashing cursor. When you press the RETURN key, the characters on the screen are read one at a time (by the subroutine at $FFCF) and placed in an 80 character input buffer for analysis. This consists of a check for a digit at the start of the line which indicates a BASIC program line – a non–digit at the start would be taken to mean that the line was a direct command.

If the line is a BASIC program line (and of course that includes a null line which indicates a line to be deleted from the program), the number at the start of the line is read and converted into an integer of two bytes. The remainder of the line has the keywords identified and converted into single byte tokens, and then the entire line is inserted into the BASIC program. In actual fact, BASIC is searched for the line number given at the start of the line. That line is then deleted from the program. If only a line number was given the routine returns to the holding loop. If the line number is followed by text, it is inserted into the program in RAM as a BASIC line. Since the remainder of the program must be "shifted" up or down in memory to accommodate the alterations, another ROM routine is called that regenerates the link addresses for the entire BASIC program.

The link address is simply an address that tells the system where the next line of BASIC text is located. Every line has a link address. This allows the interpreter to skip over lines, for example, when a GOTO is encountered. This way the interpreter can look at the line number, compare it to the target line number, and jump directly to the next line if they don't match. Imagine the time consumption if the interpreter had to look through the entire line to find the next line number. Also, when inserting or deleting lines, the editor can determine immediately how many bytes to open up or take out. Although these links consume memory, the advantages are well worth the 2 bytes.

Direct command lines (with no number at the start) also have keywords changed to tokens; however, the BASIC program pointer is altered to point to the input buffer instead of the BASIC program, and the line is then interpreted in exactly the same way as a normal program line.

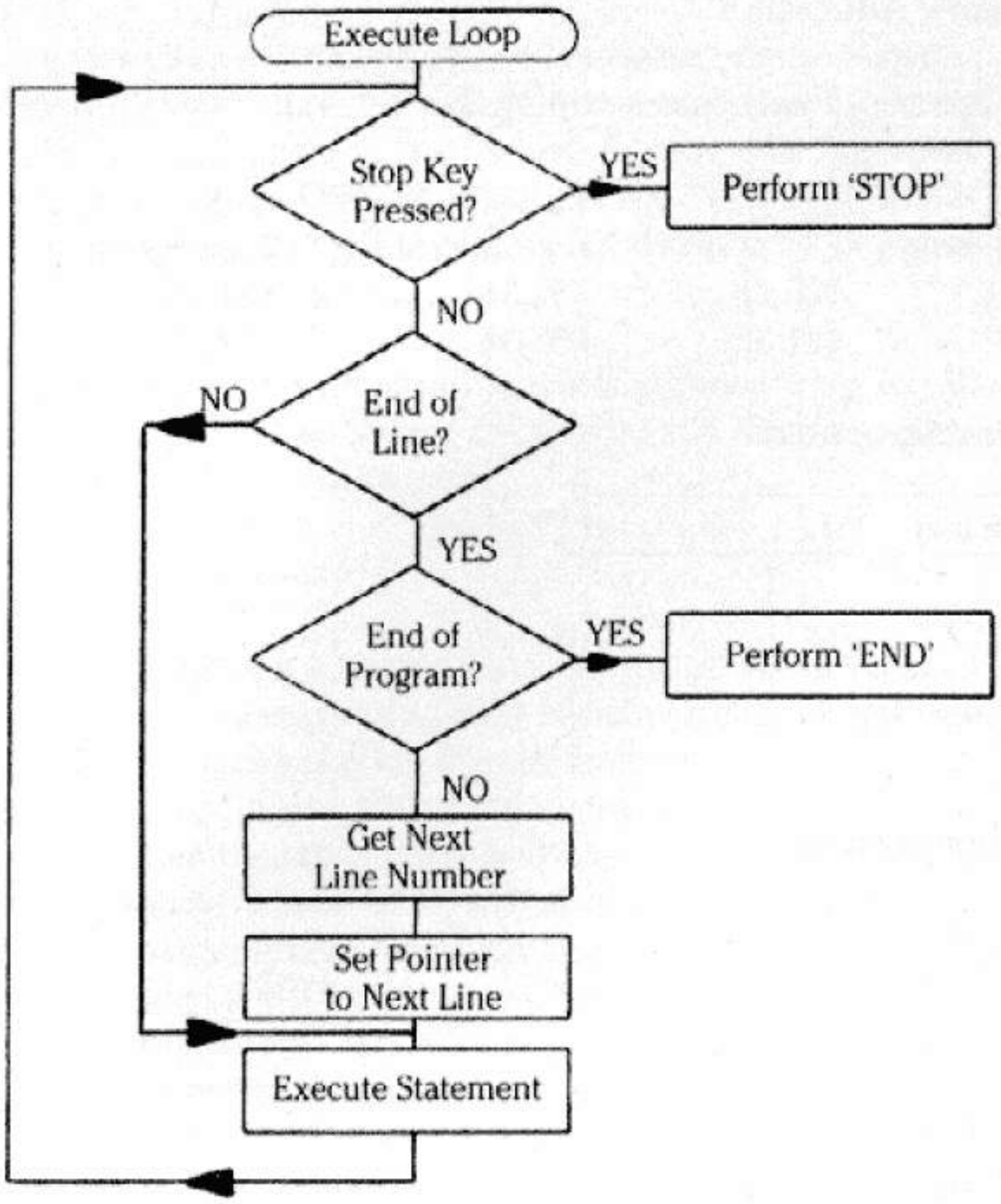
Let's return to the keyword to token conversion; this is quite a time–consuming task – one reason why it is done when the line is entered and not during program execution, is the delay during a run would be intolerable. The other principal reason is one of space saving – a one byte token saves several bytes over keywords such as "RESTORE"! All these keywords are held in a table starting near the beginning of ROM with the last character in each having bit 7 (the most significant bit) set to 1. The "crunch" routine, as it is sometimes called, scans the keyword table character by character for a match with the first character in the text. When this is found, successive characters are checked until a mismatch occurs, and if this mismatch is only bit 7 then a complete keyword has been found and is replaced in the text by a token generated from a number giving the table position of the keyword with bit 7 set. If the mismatch is other than bit 7 then the process is repeated through each keyword in the table and if no match is found, the next character in the text is used as the starting point for the next table scan.

The fact that a mismatch in bit 7 is all that is required to terminate the checking sequence gives us a clue as to why keywords can be shortened. For instance "nE" can be used instead of 'next', since the 'E' is the same as 'e' but with bit 7 set and this forces the match to me made. But, of course, the match will only be valid for the first match in the table – thus, 'next' occurs before 'new' and will be the first match for 'nE'. Similarly, 'read' occurs before 'restore' and will be recognised if "rE" is used – restore would have to be abbreviated to 'reS'.

The most important direct command which will be interpreted is the RUN command, for this is the command that instructs the interpreter to start execution of the BASIC program that you have built up. The interpreter checks to see if you have given a line number, and if you have, performs a CLR followed by GOTO the line number specified. Otherwise it resets all pointers to the start of BASIC text, activates a clear and then returns to the main interpreter which now has had its pointers reset, and execution starts at the beginning of the program.

The Interpreter

The core of the interpreter first checks if the stop key is pressed (in which case it performs the STOP command automatically). It then handles the BASIC line pointers; it checks for the end of the program (which is marked by three consecutive zero bytes) and exits if found; it also handles the occurrence of the end of a line (a single zero byte) by moving the pointers past the line number and forward pointer of the next line. The start of the current statement is saved (if not in direct mode) so that CONT will know where to pick up from if execution is stopped for any reason.



Flowchart Of The Main Execution Loop

Tokens

The statement is then interpreted. Here, the first byte of the line is checked to see if it is a token (ie. if bit 7 = 1). If it is a valid command token (and not something like SIN or THEN), bit 7 is removed, and the resulting number (which is the position of the keyword in the table) is used to access the start address of the command routine from a table of addresses in ROM directly above the keyword table. The interpreter then jumps to the appropriate routine. If the keyword is GO, then a check is made for TO (since GOTO is the only keyword that can be split into two words) and if found, the GOTO routine is entered. If the first byte is not a keyword, the interpreter assumes that a 'LET' operation is required (since LET can of course be omitted) and the LET routine is entered. Any invalid start to the line will result in the interpreter printing the SYNTAX ERROR message, flagging that an error has occurred, and then entering the main holding loop after printing READY.

Everything else that the machine does is under control of the keyword command set and consequently some of the more important commands will be examined in detail. However, all commands require the ability to retrieve a character from the BASIC program text and a routine is provided specifically for this purpose. Within this routine is a pointer which is incremented as soon as the routine is entered and then a character is retrieved from that location. If the character is a colon or a zero-byte (both of which signify the end of a statement) then the zero (Z) flag is set – if it is not a digit, the carry (C) flag is set. If the character is a space then the routine ignores it and reads the next non-space character.

Getting Characters

This routine, which starts at $0070 ($0073 in the VIC and 64), is often referred to as the CHRGET routine (pronounced "char-get") and, being placed in RAM, is easily modified to allow additional commands to be added as in the Programmers Toolkit. Other user functions can also be implemented, and a variety of techniques are used to patch into this important routine.

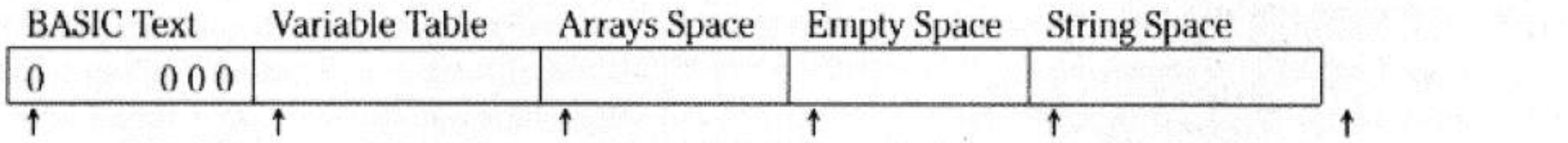
As well as using the CHRGET routine at $0070, the interpreter will often enter at $0076 ($0079 in VIC/64) instead. This entry (sometimes referred to as the REGET or CHRGOT routine) does not increment the character pointer and is used to fetch a character from text that the interpreter already GOT once and needs to GET again.

Variables

It is also worth examining how BASIC stores its variables. The simplest are the ordinary numeric variables. These are stored at the end of BASIC text, each taking seven bytes. Without describing the complex technique of "offset exponent, normalized binary floating point" storage, it is worth pointing out that all numeric operations are performed in this format. Even integer variables (stored in two byte fixed point format) are converted back to floating point whenever they are accessed. There is a floating point accumulator in RAM consisting of 6 bytes – the first is the exponent, the next 4 are the mantissa, and the 6th is the sign which is recovered from the stored form of the number. There are additional floating point accumulators which are used as workspace when evaluating expressions. The principal accumulator (FPACC#1) stores intermediate results and is also the accumulator upon which the trigonometric functions operate.

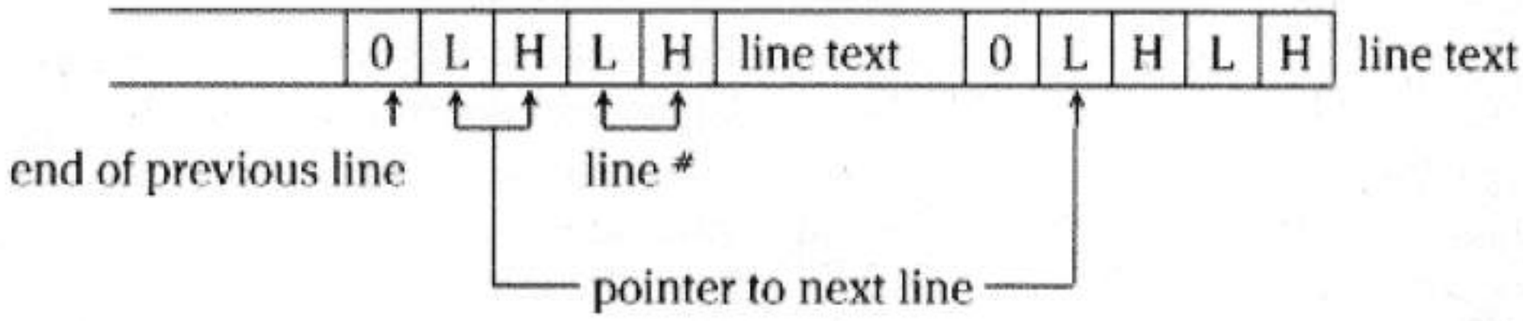
Numbers in arrays are stored in a similar format (except that integers use only two bytes instead of seven and floating point numbers five instead of seven) – but there is a header to each array which contains the array name, the number of bytes in the array and information about the dimensions of the array. However, these are stored after ordinary variables which means that every time a new ordinary variable is defined, the entire array table has to be shifted up seven bytes.

BASIC RAM Memory Allocation

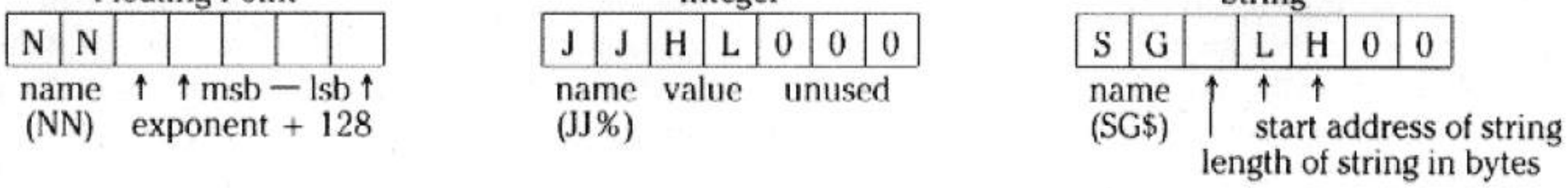


	Start of BASIC	Start of Variables	Start of Arrays	End of Arrays	Bottom of Strings	Top of Memory
BASIC 4/2:	$28,29	$2A,2B	$2C,2D	$2E,2F	$30,31	$34,35
VIC/C64:	$2B,2C	$2D,2E	$2F,30	$31,32	$33,34	$37,38

BASIC Text Line Structure



Variable Contstruction



Strings

Ordinary string variables are also stored in seven bytes, although only three are actually used for string information. These three bytes are called a string descriptor, with the first byte holding the number of characters in the string and the other two the address of the first character in the string. If a string is defined in a program (e.g. A$ = "TEST STRING"), the string is already in RAM and the descriptor will point to the characters within the text area. However, as soon as strings are manipulated they need to be stored somewhere else, otherwise a string operation could actually corrupt the BASIC program text!.

This problem is overcome by using the empty space after the arrays. But since the arrays are moved up each time a new variable is defined and of course the end of arrays will change if a new array is created, strings cannot be placed immediately after the arrays. If they were, the strings would have to be moved up as variables were created, and all string descriptors modified accordingly – a mammoth task. Instead, strings are built starting at the end of RAM. A pointer is set to the end of RAM, and whenever a string is created this pointer is moved down by the number of characters in the string and the string stored starting at that point. Unfortunately, if another string is defined, this pointer is moved down yet again, until it reaches the end of the arrays.

Garbage

When this pointer crashes into the top of the arrays, there are two alternative solutions. The first is to abort and print an OUT OF MEMORY error – a defeatist answer!. The other solution assumes that there will be strings in this area which are no longer of use – in other words the original descriptor for that string has been changed because the variable has been redefined and has therefore been rewritten elsewhere in the string storage area. Thus the string storage area contains garbage, and so a routine, referred to as "garbage collect" is invoked to weed out all those unreferenced strings. It does so as follows:

A pointer is set to the end of RAM (and so the top of string space) – all string descriptions are checked (in both ordinary and array string variables) to find which pointer is closest in value to this garbage collection pointer. The position of the final character of the string is calculated by adding its length to the start of the string and the string is moved up to fill any unused space – the string descriptor is then updated and the string space pointer moved down to the start of the string.

All string variables are then checked for the next string

descriptor below this pointer and the same process repeated until all strings have been checked and moved up to take up the space occupied by the garbage thereby releasing space for more strings. Only when the garbage collect routine fails to release enough space does the OUT OF MEMORY error occur.

In other words, the string space is scanned from top to bottom; all unused strings are deleted and all used strings moved up to overwrite them.

Minutes or Hours

Unfortunately, if a large number of strings are being used (especially if a large string array is defined) then the garbage collect routine has a lot of work to do. For instance, if the only strings in use are all the elements of a 100 element array, then the string space will have 100 strings in it and there will have to be 100 searches through all string descriptors – amounting to 10000 descriptor checks! If there are more than 1000 strings then over a million checks will have to be done as well as 1000 string moves and updates – no wonder garbage collect can take several minutes (or even hours!!!). BASIC 4 (which is supplied in the new 4000 and 8000 series PETs and is available as an upgrade for older PETs) has solved this problem by adding two bytes extra to each string; these are used by a much improved garbage collection routine to an acceptable level.

Input & Output

Before going on to consider a few BASIC commands, it is worth considering how the operating system handles input and output. There are three main types of output routines directing characters to the screen, the cassettes or the IEEE bus. All have their own separate routines, but can be accessed through a single output routine at $FFD2. This routine checks the contents of location $BO ($9A in VIC/64 – often referred to as the CMD output device number) and uses this to access the appropriate output routine. If set to device number 3, the screen output routine is used. This routine handles all ASCII characters (including clear–screen, cursor movements etc.) and places the character on the screen, updating the screen pointers.

IEEE Management

If the CMD output device is greater than 3 then the IEEE handshake routines are called, while any other value is assumed to be a cassette operation and the character is placed in the appropriate cassette buffer. The buffer pointer is incremented and if it is 192 the entire buffer contents are written to tape and the buffer reset. It is worth noting that CHR$(10) – line feed – is ignored when sent to tape to avoid it interfering with the data when is is read back.

Whenever the $FFD2 output routine is called, it is assumed that not only has $BO been set correctly, but that a channel has been opened to the cassette or IEEE bus. This ensures that IEEE or cassette protocols have been observed and the data is sent to a valid device.

Keyboard

In a similar way, there is an input routine at $FFCF which uses $AF ($99 on VIC/64) to indicate the device number from which input will be taken. If it is zero, then the keyboard is taken as the input device although strictly speaking the input is taken from the screen. As soon as the routine is entered (and $AF=0), the cursor is set flashing and the current position of the cursor logged as the start of the line (thereby avoiding any prompts being accepted as part of the input). Thereafter characters are read from the keyboard buffer and placed on the screen until the RETURN key is hit at which time the routine is left, returning the first character on the line. Subsequent calls to the routine do not set the cursor flashing, nor do they allow further input to be accepted. Instead they return successive characters on the line until the last character is read. The next call of the routine starts the process over again.

Like the output routine, the input routine assumes that all necessary protocols have been handled (normally through the OPEN command). Characters from cassette are read from the cassette buffer until such time as the buffer pointer reaches 192 at which time the next data block is read from tape. Alternatively, the IEEE routine simply handles the necessary handshaking on the IEEE bus.

Error Messages

Unfortunately, this is not the full story, since there are occasions when output to the screen has to be suppressed. A flag exists which is used to suppress screen output. This flag is at $10 ($13 in VIC/64) and contains the logical file number of the most recently accessed input and output files. Therefore if $10 is zero, both input and output are normal and this allows INPUT error messages for instance to be printed. If either is off normal (i.e. if an INPUT# or PRINT# command has just been obeyed) then these error messages are suppressed.

Although mention has been made of the INPUT routine at $FFCF, there is another routine at $FFE4 which collects a single character from the specified device. In the case of cassette or IEEE input this is no different to the usual input

routine. But for keyboard input it has the effect of GET – where only a single character is fetched from the keyboard buffer, no cursor is flashed nor are the typed characters placed on the screen.

LET Command

The first BASIC command to be considered is the LET command. As already mentioned, this is the only time when a statement does not need to start with a keyword token. Although in principle the command routine is very simple, the evaluation of the expression on the right hand side of the equal sign is extremely complex and the details cannot be described fully here. Any standard reference work on computer compilers will describe the technique of evaluating arithmetic expressions.

The first task of the LET routine is to ascertain which variable is the target variable on the left of the expression and then search the variable table or array table for it – setting $44/5 ($47/8 in VIC/64) to point to the variable. If not found, then a routine is called which creates a variable (or array) as required. In addition, a note is made at this point of the variable type.

After confirming that "=" follows the variables, the interpreter enters a routine which evaluates the expression after it. This routine is made more complex by the need to handle string expressions as well as numeric expressions.

To evaluate an expression, it can be considered as individual terms (i.e. a number, a variable or "pi") separated by operators (such as + –/*etc.). The expression is scanned for the highest priority operator – the exponentiation operator – and the terms either side of it are placed in the two main floating point accumulators. If one of the terms is a bracketed expression, the evaluation routine is re–entered to evaluate the expression in brackets. Then the exponentiation routine is called and the intermediate result saved on the stack. This is done for all operators working down from the highest priority to the lowest priority (addition) which will result in all operators being assigned their correct order of precedence.
Experts will point out that this is not precisely what happens, but it serves as an example. In fact, the process is significantly more complex than this but the fundamentals are the same.

String expressions are evaluated rather more simply, with intermediate string results being placed (in descriptor form) on a descriptor stack until a final descriptor is computed. The result is passed back as a pointer which points to the final descriptor.

In the case of numeric results, the final result will be placed in the main floating point accumulator. The result is then converted to the appropriate form for storage (e.g. the floating point result converted to an integer) and then stored at the target variable location.

FOR–NEXT

The next command to be examined is the FOR command, which together with NEXT allows a simple method of loop control. As with all other commands, FOR has its own routine and its first task is to perform the LET routine on the FOR variable. This sets the initial condition for the loop. The stack is then searched to find if there is already a FOR loop active using the same variable. If there is, then it and all FOR loops declared after it are deleted from the stack.

After checking to see if the stack can hold the FOR entry (if it cannot, an OUT OF MEMORY error is printed), the text is scanned for the start of the next BASIC statement after the FOR command and the pointer to this statement and line number are pushed onto the stack. The routine then confirms that "TO" is the next token in the text and the expression following it is evaluated. The result (the final value for the loop) is then pushed onto the stack. Next, the text is checked for the STEP token and if present, the STEP expression is evaluated. If STEP is not present the STEP value is set to 1. The sign of the STEP value is then placed on the stack followed by the absolute value (ABS) of the STEP. The pointer to the FOR variable is then pushed onto the stack and finally the FOR token ($91) is pushed on the stack as a marker.

The interpreter continues execution of BASIC text until it reaches a NEXT command. If a variable is specified, the variable table is searched for and the pointer to the variable compared with the variable pointer of all FOR entries on the stack. If not found then NEXT WITHOUT FOR is printed. If found, then the current value of the FOR variable is retrieved from the variable table, the STEP value is added and the result compared with the TO value. If greater, the FOR entry is removed from the stack, together with any GOSUB or FOR entries that were declared after the current entry and execution continues normally.

However, if the result is equal or less than the TO value, the pointer and line number are taken from the FOR entry on the stack (they point to the first statement following the FOR command) and execution of the program is resumed at that point. If the STEP is negative, it is subtracted from the FOR variable, and the loop terminates if the new value is less than the TO value.

'FOR' Stack Entry

LO	Pointer to first
HI	statement in loop
HI	Line number of first
LO	statement in loop
M4	
M3	
M2	'TO' value
M1	
EXP	
	Sign of 'STEP'
M4	
M3	
M2	'STEP' value
M1	
EXP	
HI	Pointer to
LO	'FOR' variable
$81	'FOR' Token

'GOSUB' Stack Entry

HI	Pointer to
LO	'GOSUB' statement
HI	Line Number of
LO	'GOSUB' statement
$8D	'GOSUB' Token

GOSUB & RETURN

Since it also uses the stack, GOSUB will now be described. On entry to the GOSUB routine, the stack is checked to see it it can hold the entry and then the current pointer and line number are pushed onto the stack, followed by the GOSUB token ($8D) as a marker. The routine then joins the GOTO routine which reads the line number, converts it into a two byte integer, searches for the line number in the text and then sets all the pointers to continue execution from that point.

RETURN simply checks the stack for the latest active GOSUB entry (deleting any FOR entries on the way!) and resets the pointers which then point to the middle of the GOSUB command. The DATA command routine is then entered which scans to the end of the current statement, updates the pointers accordingly and resumes execution at that point.

CONTINUE

CONT is another routine which updates the program pointer (which is the pointer held in $77/8 in the CHRGET routine). This time, however, the new value is retrieved from $3A/B ($3D/E in VIC/64) where it was stored when the program was stopped. The contents of $3B are first checked to see if it is zero – if it is then the address pointed to must be $00xx and the last command must have been a direct command (if an error occurs $3B is also set to zero).

If this is the case, the interpreter does not allow the program to continue. The reason for this is twofold. The first is that, in direct mode, the statement is being executed in the input buffer and if the statement has been stopped, it is possible that it will have been overwritten by a new line of input, in which case CONT would try to resume a potentially nonsensical statement. The other reason is that an error can occur at any point during the execution of a statement and can leave the BASIC pointers and vectors in an indeterminate state – therefore, while CONT might rejoin BASIC at the correct point, there is no guarantee that execution would continue correctly due to the possible corruption of the BASIC pointers. If CONT is allowed, the CHRGET pointer and line number are restored and execution continues from that point.

Having mentioned that CONT during direct mode could cause conflict in the use of the input buffer and is consequently disallowed, it is worth adding that this is the same reason why the INPUT or GET commands are not allowed in direct mode. In both cases, the input buffer is used to hold a string of characters which would overwrite the direct mode statement already in the buffer and cause great confusion to the interpreter!.

Pointers

Before going on to look at the final commands of SAVE and LOAD, it is worth examining briefly the pointers which indicate the various sections of memory used by the interpreter. The first is the start–of–BASIC pointer (at $28/9, $2B/C in VIC/64) which is always set to the start of BASIC text space. This address varies from machine to machine, but it always points to the first byte of the first lines' header which contains the pointer to the next line as well as the line number.

The next pointer ($2A/B, $2D/E in VIC/64) indicates the start of the variable table and normally points to the first byte following the three consecutive zero bytes indicating the end of the BASIC program. As well as indicating the start of the variable table it is also used as an end-of-BASIC pointer.

The start of the array table is held in $2C/D ($2F/30 in VIC/64), and the end in $2E/F ($31/2) which is also used to show the end of RAM usage (working up from the bottom of RAM). It is this pointer that the strings, which work down from the top of RAM, must not pass. The lower limit of strings is held in $30/31 ($33/34) and it is the comparison of this with $2E/F that instigates the garbage collect routine.

The last pointer is $34/5 ($37/8), which gives the Top of Memory. This can be lowered to allocate some memory for assembler routines for example. Note that if this is done, a CLR instruction should follow the two POKEs.

SAVE

When you use the SAVE command, the limits of RAM to be saved are indicated by the Start-of-BASIC pointer and the Start-of-Variables pointer. If the device is one of the cassettes then a header is created with the start and end addresses included together with the program name. This is then written to tape followed by the complete program. However, if an IEEE device is specified, the program name is sent first (used to open a channel in the device) and the program is sent byte by byte – the first two bytes sent being the start address of the program.

LOAD

LOAD is rather different, however, and its operation depends on whether or not you are operating from within a program or in direct mode. If from tape, the header of the specified program is searched for (if none is specified, the first program header read is used). This header contains the start and end addresses of the program and it is the fate of these which differs from program to direct mode. If an IEEE LOAD is performed, the program name is sent to the IEEE bus (to allow the device to find the program – for instance on the disk unit) and the first two characters received are taken as the start address of the program. When program LOAD is complete the IEEE bus signals the fact and the address of the last byte loaded used as the end address, since this address is not explicitly saved as part of the program.

The VERIFY command operates identically to LOAD, except that bytes are compared with those in RAM rather than being stored.

In direct mode (and that includes a LOAD using the shifted RUN/STOP key), once the program is loaded, the start-of-variables pointers is set to the end address of the loaded program. In order to avoid problems of trying to read variables from a now corrupted variable (and array) table, a CLR is performed (which sets the start of arrays and end of arrays to the end of program pointer and resets the lower limit of string storage to the upper limit of RAM). This is followed by the recalculation of all the link addresses at the beginning of each line of the BASIC program. Finally, the main waiting loop is rejoined at the point at which READY is printed.

In program mode, LOAD operates rather differently. The actual mechanism remains the same, but the End of BASIC/Start of variables pointer is left untouched. This allows the newly loaded program to access the same variables as the program which loaded it. However, if the loaded program is longer than the calling program, the variable table will be over-written by the end of the new program and cause no end of problems. This is why programs loaded from within a BASIC program must always be shorter than the calling program.

Once loaded, BASIC is reset to continue execution at the beginning, and a partial CLR performed which cleans up the stack to remove any outstanding GOSUB and FOR entries and executes the RESTORE command routine. Unlike LOAD in direct mode, LOAD in a program does not perform the regeneration of the link addresses.

. . .And Beyond

That then is our look inside BASIC – or at least as much as time and space will allow. Further investigations can be made with the use of a simple disassembler (which converts the machine code back into assembly language, but without labels or variable names). With the information given above, and the details given in tables of ROM and workspace RAM, most readers should be able to delve a little deeper under the bonnet. Armed with a deeper understanding of the software processes involved in the BASIC interpreter, many users have been able to make their Commodore computers sing and dance far in excess of what their creators ever imagined!

Messing With The Stack

Garry G. Kiziak
Burlington, Ont.

Sometimes it would be nice to be able to leave a subroutine by a GOTO statement rather than the usual RETURN statement. Doing so is usually considered 'poor programming practice'. However, there are times when it can be useful (e.g. implementing an 'escape key' feature in an input subroutine).

The most pressing reason why one should avoid this programming technique, aside from the fact that it encourages 'bad programming habits', is that it does terrible things with the stack.

Try the following program.

```
100 gosub 200
110 print "eureka, it works!"
120 end
200 if i=50 then return
210 i=i+1
220 print i;" . here i am again."
230 goto 100
```

Program 1.

The program will print out the message "HERE I AM AGAIN." twenty three times and then stop with an "? OUT OF MEMORY ERROR IN 200".

In fact there is plenty of memory available. What has happened is that each time line 100 is executed, 7 bytes are placed on the stack (5 of these bytes tell the computer where to return to when it encounters the 'RETURN' statement and the other two bytes are placed there for internal reasons).

Since the 'RETURN' statement in line 200 doesn't get executed until I=50, the stack quickly fills up (the stack can contain a maximum of 256 bytes, but much of that is used by the operating system). When it is full the 'OUT OF MEMORY' error occurs.

Fortunately there is a simple way out of this problem, simply add the following line.

```
225 sys clear : rem where clear = 46610 on 4.0 PETs
                              and   50583 on 2.0 PETs
```

This 'SYS' command accesses a ROM routine which is part of the 'CLR' command and effectively clears the stack each time it is called. Now since the stack is no longer allowed to fill up, the program should run to its conclusion printing 'HERE I AM AGAIN.' 50 times followed by 'EUREKA IT WORKS.'.

The CLEAR address for the Commodore 64 is 42622, and for the VIC 20 it is 50814. However, as is well known (See The Transactor Vol.4 Issue 3 page 26), it no longer works. This is somewhat puzzling, especially when you look at the disassembled machine code on each of the machines and find that they are identical – except for some address changes which are necessary because the routines are located in different places on each machine.

Here is the disassembled code on the Commodore 64.

```
01 $a67e 68    clear pla       ;save the 2 bytes most recently
02 $a675 a8          tay       ;pushed onto the stack
03 $a680 68          pla
04 $a681 a2 fa       ldx #$fa  ;place $fa in the stack pointer
05 $a683 9a          txs
06 $a684 48          pha       ;restore the two bytes saved
07 $a685 98          tya       ;above
08 $a686 48          pha
09 $a687 a9 00       lda #$00  ;do some house cleaning
10 $a689 85 3e       sta oldtxt+1
11 $a68b 85 10       sta subflg
12 $a68a 60          rts       ;return to the command interpreter
```

Before I explain how this works, let me recall one important fact about the 6502 (or 6510) processor, and how it works in conjunction with the JSR statement. Namely, that each time a JSR statement is executed, the processor automatically places two bytes onto the stack. These two bytes tell the processor where it should continue executing instructions from when it encounters an RTS statement. On executing the RTS statement, these two bytes are

removed from the stack and execution continues from that address – actually that address plus one.

Here is how the CLEAR routine works. First it saves the last two bytes that were pushed onto the stack in the A and Y register (lines 1 to 3). This is necessary because this routine is normally called by a JSR command in the 'command interpreter'. These two bytes therefore tell the processor where to return to. (On a Commodore 64 these two bytes are $A7 and $E9 so the return address is $A7EA). Lines 4 to 5 place $FA in the stack pointer. This in effect clears the stack since this is the value put there by the computer on power up. Lines 6 to 8 replace the two bytes saved above so that the RTS in line 12 can return control to the command interpreter.

Since this code is virtually identical on all machines, why doesn't it work on the Commodore 64 or the VIC 20. Well, the problem doesn't lie in this routine. Instead it lies in the routine that executes the 'SYS' command.

An extra feature has been incorporated into the 'SYS' command on the Commodore 64 and the VIC 20, namely the ability to set the A, X, Y, and status registers before entering the actual machine language routine. These registers are set by poking values into locations 780–783 before SYSing to the required routine. In order to accommodate this extra feature the code that executes the SYS command must be different. Here's what it looks like on the Commodore 64.

```
01 $e12a 20 8a ad jsr   frmnum ;get address after sys command
02 $e12d 20 f7  b7 jsr  getadr ;convert it to an integer
03 $e130 a9 e1     lda  #$e1   ;put address $e147-1
04 $e132 48        pha         ;on the stack so rts in
05 $e133 a9 46     lda  #$46   ;user's routine will continue
06 $e135 48        pha         ;execution at $e147
07 $e136 ad 0f  03 lda  $030f  ;get status register from
08 $e139 48        pha         ;783 and save on stack
09 $e13a ad 0c 03  lda  $030c  ;get a-register from 780
10 $e13d ae 0d 03  ldx  $030d  ;get x-register from 781
11 $e140 ac 0e 03  ldy  $030e  ;get y-register from 782
12 $e143 28        plp         ;get status reg back from stack
13 $e144 6c 14 00  jmp  (linnum) ;jump to the user's routine
14 $e147 08        php         ;save status register on stack
15 $e148 8d 0c 03  sta  $030c  ;store a-register in 780
16 $e14b 8e 0d 03  stx  $030d  ;store x-register in 781
17 $e14e 8c 0e 03  sty  $030e  ;store y-register in 782
18 $e151 68        pla         ;get status register back
19 $e152 8d 0f  03 sta  $030f  ;store it in 783
20 $e155 60        rts         ;return to command interpreter
```

Notice that the A, X, Y, and status registers are loaded from locations 780–783 (lines 7 to 12) before the actual machine language routine is entered (line 13). Also notice that two extra bytes are pushed onto the stack before entering the M/L routine as well (lines 3 to 6). These are put onto the stack so that when an RTS is encountered in the M/L routine, control will return to $E147 where the A, X, Y, and status registers are put back in locations 780–783 before returning to the command interpreter.

These extra two bytes are the cause of all our problems. For when the CLEAR routine is called by SYS CLEAR, these two bytes are saved instead of the two bytes that will return it to the command interpreter. End result? When the RTS is encountered in the CLEAR routine, control returns to $E147, and the stack is clear. Now when the RTS is encountered in $E155 of the SYS routine, there is no legitimate return address on the stack, so it takes two meaningless bytes and returns to heaven knows where.

If you understood the above, the solution to the problem is now quite simple. Instead of SYSing directly to the ROM routine, SYS to your own routine where you first remove those two extra bytes from the stack and then jump to the ROM routine.

For the Commodore 64	For the VIC 20
PLA	PLA
PLA	PLA
JMP $A67E	JMP $C67E

On the Commodore 64, the following BASIC loader added to the beginning of Program 1 will make that program work properly.

```
10 clear=828 : for k=clear to clear+4 : read j : poke k,j : next k
20 data 104,104,76,126,166
```

On a VIC 20, replace line 20 with:

```
20 data 104,104,76,126,198)
```

Of course you must still include line 225

```
225 sys clear
```

Notice that the routine is completely relocatable, so you can put it in any (safe) place that you like.

POP For The Commodore 64

The CLEAR routine does its job just fine. However, it may also do more than you really want. Its purpose is to clear the entire stack – of RETURN addresses, of FOR . . . NEXT loop pointers, and anything else that may be on the stack. Yet there may be times when all you want to do is 'POP' the last RETURN address off the stack. The following routine will do just that on the Commodore 64.

```
10 pop=828 : for k=pop to pop+24 : read j : poke k,j : next k
20 data 104,104,169,255,133,74,32,138,163,154,201,141,240,5
30 data 162,12,76,55,164,104,104,104,104,104,96
```

Replace lines 10 and 20 of the program above with these two lines and replace line 225 with:

```
225 sys pop
```

You will see that the program works just as well. This time however the entire stack is not cleared, just the last RETURN address. Also like the normal RETURN statement, any active FOR . . . NEXT loops within the subroutine will be removed by the POP. Notice that this routine is also relocatable so it can be placed in any 'safe' place.

The Un-Token Twin's

Richard Evers

The purpose of the 'un-token' twin's is to supply you with a unique method to list programs from memory, or from disk. Normally this task would be considered a rather anti-climatic event, but the programs demonstrate how one can make use of information already living inside your machine.

Similar programs released in the past have usually kept their keyword table in a stack of data statements somewhere within the program. This fact alone has made these programs quite large in size and fairly painfull to key in. My program uses the basic ROM table of keywords for its token/keyword conversion. With this change of direction, a lot of space has been saved in memory, and a fair bit of time and trouble trimmed off the keying in procedure. Hopefully, less code will mean less mistakes.

As a bonus, the link address for the following line is printed before the line number. If you find this totally useless to you, simply remove it. . . the link address is held in variable 'C'.

As the program progresses, it does a number things. The first is to actually get the character to be printed. The value is checked to see if it is a quote. If so, the program flow will change direction a tad to avoid any trouble with tokenization of capital letters and reverse case characters. If not, the program will plod ahead to check if the value encountered was a keyword. If so, 128 is subtracted from the ASCII value, and an keyword array variable is assigned to it. In this manner, all of the keywords match up exactly with their token values.

After this, if the value was not a keyword, but the 'in quotes' flag has been set, then the value is OR'd with 64 to get rid of any strange reverse case characters that might be pretending to be control characters. Control characters, or pseudo control characters quite often make a real mess out of the screen when they are printed. The 'quick OR with 64' will reduce the chances of this happening. When all of this has been performed, the character is finally printed to the screen, and a test or two is done to keep the flow going.

Notice the line numbers in each version. The disk version has been written with successive line numbers because it will probably be alone in memory. The RAM version has somewhat larger line numbers. Since this program must co-exist with the program you wish to un-tokenize, the larger line numbers were chosen to make room for the other program. Hopefully it will not have line numbers that will interfere with 'un-token memory'.

For now, that is all that is to be said about the 'un-token twins'. If you find further methods in which to make them even better, please send us a note telling of your technique. We are always looking for reader support, and will not hesitate to print what is sent to us if given the chance. Whenever you feel creative, please keep us in mind.

```
10 rem * un-token memory - richard evers
15 rem * will list a basic program in memory
20 rem ********************************
25 rem * 4.0 basic ts=45234 : te=45579 : sb=1025
30 rem * c64       ts=41118 : te=41373 : sb=2049
35 rem * ts + te = start and end of rom keyword table
40 rem * sb = start of basic
45 rem ********************************

63900 print chr$(147); : clr : ts=45234 : te=45579
      : sb=1025 : ps=sb : rem * 4.0 basic *
* set up your variables for your particular machines here

63901 dim kw$(90) : kw=0 : for a=ts to te : k=peek(a)
* the array kw$(90) will hold all keywords, the loop will
  bring them in

63902 if k<128 then kw$(kw)=kw$(kw)+chr$(k) : next
      : print chr$(147) : goto63904
* if the value peeked was below 128 ascii, then it is just part
  of keyword

63903 k=k-128 : kw$(kw)=kw$(kw)+chr$(k) : print
      kw$(kw), : kw=kw+1 : next : print chr$(147)
* end of token encountered - subtract 128, add to string and
  adjust pointers

63904 c=peek(sb)+256*peek(sb+1) : d=peek
      (sb+2) + 256*peek(sb+3) : if c=0 then 63910
* 'c' and 'd' hold the link address and line# respectively

63905 print c;d; : for e=sb+4 to c-2 : f=peek(e)
      : f$=chr$(f) : if qt then 63907
* print 'c' and 'd', then loop throughout the entire line to
  print

63906 if f>127 then f$=kw$(f-128)
* if keyword encountered, subtract 128 then assign array
  variable to it
```

```
63907 if qt then f$ = chr$(asc(f$)or64)
* if quote flag set, 'or' the value with 64 to get rid of control
  characters

63908 print f$; : if f = 34 then qt = not qt
* print the string, check if the value was a quote and flip
  QT if so

63909 next : print : qt = 0 : sb = c : goto 63904
* next the loop, drop a line and re-adjust pointers to
  continue on

63910 print : print " program size " sb-ps " bytes " : end
* program complete ! show the size of program and end
```

```
10 rem * un-token disk - richard evers
15 rem * will list most basic programs from disk
20 rem ********************************
25 rem * 4.0 basic ts = 45234 : te = 45579 : sb = 1025
30 rem * c64       ts = 41118 : te = 41373 : sb = 2049
35 rem * ts + te = start and end of keyword table in rom
40 rem * sb = start of basic
45 rem ********************************

50 print chr$(147); : clr : ts = 45234 : te = 45579
    : sb = 1025 : qt = 0
* this one has been set up for 4.0 basic - re-adjust to suit
  your needs

55 dim kw$(90) : kw = 0 : for a = ts to te : k = peek(a)
* array kw$(90) will hold the basic keywords

60 if k<128then kw$(kw) = kw$(kw) + chr$(k) : next
    : goto 70
* this checks if end of keyword has been encountered

65 k = k-128 : kw$(kw) = kw$(kw) + chr$(k)
    : print kw$(kw), : kw = kw + 1 : next
* value is beyond 128 ascii so the end of keyword has come
  - act accordingly

70 print chr$(147) + " program file name " ; : input pn$
    : if len(pn$)>16 then 70
75 input " drive number " ;d$ : if d$< " 0 " or d$> " 1 "
    then 75
* routine to get the program name and drive number
  - nothing special here

80 print chr$(147); : open5,8,5, " " + d$ + " : " + pn$ + " "
* open the file on drive # d$ with file name pn$
```

```
85 gosub 140 : ps = asc(a$) : gosub 140
    : ps = ps + 256*asc(a$)
* get the low and high bytes of the start address from disk

90 if sb<>ps then print " not a basic program "
    : close 5 : end
* if the address is higher or lower than basic as set, reject
  and end

95 gosub 140 : c = asc(a$) : gosub 140
    : c = c + 256*asc(a$) : if c = 0 then 150
* get the low and high bytes of the link address from disk

100 gosub140 : d = asc(a$) : gosub 140
     : d = d + 256*asc(a$)
* get the low and high bytes of the line number from disk

105 print c;d;
* print the link address followed by the line number - adjust
  to your taste

110 gosub 140 : z = asc(a$) : if qt then 120
* main routine to get the contents of the line - is quote set,
  skip ahead

115 if asc(a$)>127 then a$ = kw$(asc(a$)-128)
* obviously out of quotes, is it above 127 ascii ? - if so, it is
  a keyword

120 if qt then a$ = chr$(asc(a$)or64)
* if within quotes, 'or' the value with 64 to rid it of control
characters

125 print a$; : if z = 34 then qt = not qt
* print the value and flip the quote's flag if the char was a
  quotation mark

130 sb = sb + 1 : if sb + 4<c then 110
* increment the start of basic pointer and go for more is we
  haven't hit top

135 print : qt = 0 : sb = c : goto 95
* drop a line, re-adjust the pointers and continue at the top
  again

140 get#5, a$ : a$ = left$(a$ + chr$(0),1)
    : if st = 0 then return
* a one line method to get a character from disk and check if
  st set

145 close 5 : print : print " program size " sb-ps " bytes "
    : end
* close the file, display the size of program and end
```

Merging BASIC Programs

Glen Pearce
Randburg, South Africa

Several Merging techniques have been published in the past, but Glen's program eliminates the need for LISTing the program to a tape or disk file in order to prepare it for this operation. – M.Ed

I'm sure most of you have at one time or another been faced with the need to "merge" one or more useful subroutines into a BASIC program that you are writing. We all build up a library of routines over the years and inevitably have to type the whole routine in again each time we wish to use it in a new program. With program development time becoming as expensive as it has lately, I decided to write a program which would automatically "merge" a program stored on disk with the one currently held in the computer's memory – after all, that's what computers are for, aren't they – to take the drudgery out of our lives?

I have supplied two merge programs – one for the 64 and one for the professional range of Commodore computers (the 4000 and 8000 series). To simplify the entry of the merge programs, I have coded the machine–code part of the programs in DATA statements which are simply 'POKEd' into memory by the BASIC section of the program.

The machine–code merge program resides near the top of memory. As the CBM operating system also uses this area to store variables, we will have to protect the merge program in some way. Here, the ever–friendly Commodore operating system comes to our aid. By simply changing the value of 2 locations in memory, we can lower the top of memory pointers by however much we like. The CBM will then regard the address specified in these locations as the new Top of Memory and ignore any memory above that address, thus protecting the machine–code merge program. Looking at the Commodore 64 program below, the two locations are 55 and 56. Location 56 holds the HI–byte address of the top of memory and location 55 holds the LO–byte. Don't worry, this is'nt as complicated as its sounds. By multiplying the 'peek' of location 56 (HI–byte) by 256 and ADDING to it the value in location 55 (LO–byte), we arrive at the memory location that is the TOP OF MEMORY.

Now, every time we reduce the value in location 56 by ONE, we lower the top of memory by 256 bytes (remember – 56 holds the HI–byte value). We'll forget about changing location 55 for the time being – suffice to say that, by reducing value in 55 by ONE, the top of memory will be lowered by 1 byte.

NOTE: THE ACTUAL MEMORY LOCATION SPECIFIED BY THE ADDRESS IN 55 IS ALSO PROTECTED.

The statement lowering the Top of Memory pointers should always be the FIRST statement in your program and the two 'pokes' must be followed by a 'CLR' command as this instructs the operating system to bring other memory pointers into line with the Top of Memory pointers.

Anyway, back to the merge program. While the machine–code program in the DATA statements is being POKEd into memory, a running total of the values being POKEd is kept. Should the total value of the DATA statements NOT equal that in line 140, the program will be aborted with the message 'CHECKSUM ERROR' meaning that you keyed in one (or more) of the data statements incorrectly. If all goes well, a message on how to use the merge program will be displayed and the BASIC program will be erased by the 'NEW' command in line 175 (so ensure you've SAVEd the program away before running it!). The machine–code merge program will remain available for use until power–down or the top of memory pointers are changed.

To merge two programs together proceed as follows:–

a) LOAD and RUN the Merge program listed below.

b) LOAD the first BASIC program into memory in the normal manner;

c) Type SYS 32000, and press the RETURN key;

d) Type 'Y' in response to the prompt 'IS 1ST PROGRAM LOADED?'

e) Type in the filename of the program you wish to merge with the 1st one you loaded and press RETURN. If you're using a Dual disk drive, you should specify the drive number on which the program resides as well (eg. 1:PRGNAME);

f) The merge will then be performed.

If all goes well, the message 'MERGE OK' will appear on the screen once the merge is completed and the complete merged program will now reside in the computer's memory (whereupon you may merge further programs into it if you wish by simply repeating the steps from (b) through (d) above).

If an error is detected (eg: file not found, Read error, etc.), the message 'MERGE ABORTED' will be printed on the screen.

NOTE: An important point to remember is that any line numbers which are duplicated in the two programs being merged will cause the line in the SECOND program to overwrite the line in the FIRST program.

A useful feature of the merge program is its ability to merge programs written on different models of Commodore computer (this is normally impossible as BASIC programs are saved from different addresses on the different models).

```
100 rem merge 4.0
110 poke 53, 125 : poke 52, 0 : clr
120 for j = 32000 to 32476 : read x
130 poke j, x : ch = ch + x : next
140 if ch <> 51230  then print " checksum error " : end
150 print " merge basic programs - basic 4.0
160 print " load the first program into ram
170 print " type sys 32000 and follow instructions
175 new :rem caution - save before running
180 data 169,   0, 133, 209, 169, 147,  32, 210
190 data 255, 162,   0,  32,  79, 126,  32, 228
200 data 255, 201,  89, 240,   7, 201,  78, 208
210 data 245,  76, 255, 179, 162,  73,  32,  79
220 data 126,  32, 207, 255, 201,  20, 240,  18
230 data 201,  13, 240,  14, 166, 209, 224,  16
240 data 240, 231, 157, 131,   2, 232, 134, 209
250 data 208, 231, 166, 209, 240, 219, 169,  44
260 data 157, 131,   2, 232, 169,  80, 157, 131
270 data   2, 232, 134, 209, 169,  13,  32, 210
280 data 255, 169,  13, 133, 210, 133, 211, 169
290 data   8, 133, 212, 169,   2, 133, 219, 169
300 data 131, 133, 218,  32,  99, 245, 162,  13
310 data  32, 175, 247,  32,  57, 126,  32,  57
320 data 126,  32, 228, 255, 208,  24,  32, 228
330 data 255, 208,  22, 162, 103,  32,  79, 126
340 data 169,  13,  32, 226, 242,  32, 204, 255
350 data  32, 233, 181,  76, 255, 179,  32,  57
360 data 126,  32,  57, 126, 133,  17,  32,  57
370 data 126, 133,  18, 160,   0,  32,  57, 126
380 data 153,   0,   2, 240,   3, 200, 208, 245
390 data 200, 152,  24, 105,   4, 133,   5,  32
400 data 163, 181, 144,  68, 160,   1, 177,  92
410 data 133,  32, 165,  42, 133,  31, 165,  93
420 data 133,  34, 165,  92, 136, 241,  92,  24
430 data 101,  42, 133,  42, 133,  33, 165,  43
440 data 105, 255, 133,  43, 229,  93, 170,  56
450 data 165,  92, 229,  42, 168, 176,   3, 232
460 data 198,  34,  24, 101,  31, 144,   3, 198
470 data  32,  24, 177,  31, 145,  33, 200, 208
480 data 249, 230,  32, 230,  34, 202, 208, 242
490 data  32, 255, 181,  32, 182, 180,  24, 165
500 data  42, 133,  87, 101,   5, 133,  85, 164
510 data  43, 132,  88, 144,   1, 200, 132,  86
520 data  32,  80, 179, 165,  17, 164,  18, 141
530 data 254,   1, 140, 255,   1, 165,  46, 164
540 data  47, 133,  42, 132,  43, 164,   5, 136
550 data 185, 252,   1, 145,  92, 136,  16, 248
560 data  32, 255, 181,  32, 182, 180,  76, 113
570 data 125,  32, 228, 255, 166, 150, 240,  13
580 data  32, 204, 255, 162, 114,  32,  79, 126
590 data 104, 104,  76, 128, 125, 170,  96, 189
600 data  91, 126, 240,   6,  32, 210, 255, 232
610 data 208, 245,  96,  13,  13,  18,  71,  65
620 data  80,  32,  80,  82,  79,  71,  82,  65
630 data  77,  32,  77,  69,  82,  71,  69,  32
640 data  45,  32,  71,  65,  32,  80,  69,  65
650 data  82,  67,  69,  32,  45,  32,  56,  51
660 data  48,  53,  50,  50, 146,  13,  13,  73
670 data  83,  32,  49,  83,  84,  32,  80,  82
680 data  79,  71,  82,  65,  77,  32,  76,  79
690 data  65,  68,  69,  68,  63,  32,  40,  89
700 data  47,  78,  41,   0,  13,  13,  68,  82
710 data  73,  86,  69,  35,  32,  38,  32,  50
720 data  78,  68,  32,  80,  82,  79,  71,  82
730 data  65,  77,  32,  78,  65,  77,  69,  63
740 data  32,   0,  13,  13,  77,  69,  82,  71
750 data  69,  32,  79,  75,   0,  13,  13,  77
760 data  69,  82,  71,  69,  32,  65,  66,  79
770 data  82,  84,  69,  68,   0
```

```
100 rem merge c64
110 poke 56, 125 : poke 55, 0 : clr
120 for j = 32000 to 32467 : read x
130 poke j, x : ch = ch + x : next
140 if ch <> 51230  then print " checksum error " : end
150 print " merge basic programs - commodore 64
160 print " load the first program into ram
170 print " type sys 32000 and follow instructions
175 new :rem caution - save before running
180 data 169,   0, 133, 183, 169, 147,  32, 210
190 data 255, 162,   0,  32,  79, 126,  32, 228
200 data 255, 201,  89, 240,   7, 201,  78, 208
210 data 245,  76, 134, 227, 162,  73,  32,  79
220 data 126,  32, 207, 255, 201,  20, 240,  18
230 data 201,  13, 240,  14, 166, 183, 224,  16
240 data 240, 231, 157,  60,   3, 232, 134, 183
250 data 208, 231, 166, 183, 240, 219, 169,  44
260 data 157,  60,   3, 232, 169,  80, 157,  60
270 data   3, 232, 134, 183, 169,  13,  32, 210
280 data 255, 169,  13, 133, 184, 133, 185, 169
290 data   8, 133, 186, 169,   3, 133, 188, 169
300 data  60, 133, 187,  32, 192, 255, 162,  13
310 data  32, 198, 255,  32,  57, 126,  32,  57
320 data 126,  32, 228, 255, 208,  24,  32, 228
330 data 255, 208,  22, 162,  94,  32,  79, 126
340 data 169,  13,  32, 195, 255,  32, 204, 255
350 data  32,  89, 166,  76, 134, 227,  32,  57
360 data 126,  32,  57, 126, 133,  20,  32,  57
370 data 126, 133,  21, 160,   0,  32,  57, 126
380 data 153,   0,   2, 240,   3, 200, 208, 245
390 data 200, 152,  24, 105,   4, 133,  11,  32
400 data  19, 166, 144,  68, 160,   1, 177,  95
410 data 133,  35, 165,  45, 133,  34, 165,  96
420 data 133,  37, 165,  95, 136, 241,  95,  24
430 data 101,  45, 133,  45, 133,  36, 165,  46
440 data 105, 255, 133,  46, 229,  96, 170,  56
450 data 165,  95, 229,  45, 168, 176,   3, 232
460 data 198,  37,  24, 101,  34, 144,   3, 198
470 data  35,  24, 177,  34, 145,  36, 200, 208
480 data 249, 230,  35, 230,  37, 202, 208, 242
490 data  32,  99, 166,  32,  51, 165,  24, 165
500 data  45, 133,  90, 101,  11, 133,  88, 164
510 data  46, 132,  91, 144,   1, 200, 132,  89
520 data  32, 184, 163, 165,  20, 164,  21, 141
530 data 254,   1, 140, 255,   1, 165,  49, 164
540 data  50, 133,  45, 132,  46, 164,  11, 136
550 data 185, 252,   1, 145,  95, 136,  16, 248
560 data  32,  99, 166,  32,  51, 165,  76, 113
570 data 125,  32, 228, 255, 166, 144, 240,  13
580 data  32, 204, 255, 162, 105,  32,  79, 126
590 data 104, 104,  76, 128, 125, 170,  96, 189
600 data  91, 126, 240,   6,  32, 210, 255, 232
610 data 208, 245,  96,  13,  13,  18,  71,  65
620 data  80,  32,  80,  82,  79,  71,  82,  65
630 data  77,  32,  77,  69,  82,  71,  69,  32
640 data  45,  32,  71,  65,  32,  80,  69,  65
650 data  82,  67,  69,  32,  45,  32,  56,  51
660 data  48,  53,  50,  50, 146,  13,  13,  73
670 data  83,  32,  49,  83,  84,  32,  80,  82
680 data  79,  71,  82,  65,  77,  32,  76,  79
690 data  65,  68,  69,  68,  63,  32,  40,  89
700 data  47,  78,  41,   0,  13,  13,  50,  78
710 data  68,  32,  80,  82,  79,  71,  82,  65
720 data  77,  32,  78,  65,  77,  69,  63,  32
730 data   0,  13,  13,  77,  69,  82,  71,  69
740 data  32,  79,  75,   0,  13,  13,  77,  69
750 data  82,  71,  69,  32,  65,  66,  79,  82
760 data  84,  69,  68,   0
```

An Introduction To Tools & Techniques For Machine Language Programming

Phil Honsinger
Kitchener, Ont.

The special lure of writing machine language programs or subroutines is the fantastic speed that can be attained in program execution.

We can do things with machine language programs that sometimes just cannot be done in BASIC. For example, I have a digital music synthesizer keyboard hooked up to my 64 through the user port. I am using it for real-time playing of music, but for the system to react fast enough to my playing on the keyboard, I had to use machine language. The BASIC prototype program would miss too many notes if I played too fast.

Writing in machine language brings with it new problems of program design and testing. Since we are operating at a much more detailed level, we must make sure that we dot our I's and cross our T's (so to speak) or the programs just simply will not work. When there is a bug (a mistake) in a machine language program, it can sometimes be VERY difficult to find and correct, and the frustration of seeing dawn appear, and your program STILL not working right, is unbelievable.

Fear not, oh brave souls! There are many tools and techniques that can be borrowed from the professional programmer's arsenal of tricks that will help us write code that is more accurate from the start, and that will help us to find and correct our bugs much faster. As a professional programmer, I have many tricks-of-the-trade. Here are some that we can use at home:

- reference manuals
- an ASSEMBLER program
- a MONITOR program
- structured programming and testing techniques
- other programs and programmers

Reference Manuals

The 64 uses a 6510 micro-processor chip that is functionally equivalent to the 6502 chip, and there are lots of books on the market for 6502 programming in machine language. If you are an accomplished programmer and have worked with machine language before, the books for you are the more clinical 'this is how each instruction works' type of book. If you are just starting out, then some of the books are designed as a teaching tool that will step you through the language and its syntax, with plenty of examples. You could even steal programs from the books for (dare I mention it) the APPLE (6502 chip again). The APPLE programs would have to have addresses, etc. converted to 64'ese, and this area is where a good reference book will be priceless. If you can get the TRANSACTOR's special reference issue (Vol. 4, Issue 05) – everything you wanted to know, including a list of Commodore related programming books.

An ASSEMBLER

An ASSEMBLER is actually another program that allows you to develop the machine language programs in the symbolic assembler language. The assembler will then translate the symbolic program into the executable machine language code that the 6510 chip understands. This abstractness is what makes the assembler so valuable. You deal in the higher level syntax, and this means the programs will be easier to develop (and understand) and this translates into faster program development.

I have been using the PROGRAMMER's TOOLBOX from PRO-LINE software. Among other things, this software package has a very good assembler called PAL. After this introduction I will go into more detail on Assemblers, or more specifically, The Two-Pass Assembler.

A MONITOR Program

A MONITOR is another program that you will use for testing your machine language programs. It allows you to interrupt the execution of the machine language program at any point and examine the 6510s registers, check memory locations for proper results, and even manipulate data for the express purposes of testing your application program. I have been using Jim Butterfield's SUPERMON.

Structured Programming and Testing Techniques

This is a very big topic among data processing managers. Programmers who use these techniques will develop programs faster and more accurately than those who don't.

What is this mysterious subject, you ask? Well, simply put, it is a way of thinking. You design your programs so that the more general logic can be written and tested functionally before you get deeper and deeper into the dark details of the important processing. You can code and test file opening, memory location initialization, or even a cursor positioning routine as units that are somewhat "stand–alone". Once you are satisfied that these units are working correctly on their own, then they can be incorporated into the larger program, and forgotten about until changes are required in these functional areas. You could then re–test the unit with the changes on its own, and/or back in the main program.

There are, of course, performance trade–offs to this kind of programming. Subroutine calls require processing overhead by the micro–processor, and you have to decide if the design is appropriate to the application. A real–time, high speed game would need as little code as possible to keep the program reaction time to data changes as fast as required.

Other Programmers and Programs

Steal as much knowledge and experience as you can. Join computer clubs, read books and mags (like The Transactor), and experiment.

Above all don't forget to "SAVE OFTEN", and have fun.

The Two–Pass Assembler

Working directly with machine language code is a very demanding and arduous task. All you have to work with is hexadecimal numbers. To assist the programmer in generating machine language programs, special programs called ASSEMBLERS have been written.

An assembler is a program that accepts mnemonics and parameters as input, and generates as output the machine code program that the mnemonics represent.

The input coding is called the SOURCE code. It is the input data to the assembler.

The output generated by the assembler is called the OBJECT code. This is the machine language code that will actually be executed at some later time. It is the actual machine language program that the source code symbolically represents. There may also be additional output from the assembler when it generates the object code, such as something called a SYMBOL TABLE.

So as not to contradict its own purpose, the assembler is usually another machine language program. Usually an assembler will have been coded in it's own source code format, so we can use one version of the assembler to produce the next version (enhancements).

It is interesting to learn how an assembler actually generates the object code from the source code, and that's the topic of this part of the article. I will describe, in general terms, something called a Two–Pass Assembler.

A Two–Pass Assembler does exactly that. . . two passes of the source code. The first pass is used to build something called the symbol table. The second pass uses the source code and the symbol table generated in the first pass to produce the final result. . . your object code.

The object code can be stored in many ways: directly in memory (at the location that YOU tell it to); as a file on a disk drive to be loaded in memory and executed later; or even just to the printer for checking. The last option would be used when you are in the design stage of a project . . . like LISTing a BASIC program to your printer that you haven't actually RUN yet.

Now for a quick description of what is contained in the source code. . ..

OPCODES and OPERANDS	(LDA #$00, for example) (also called MNEMONICS and PARAMETERS)
LABELS	(to symbolically represent an address or the location of an instruction)
COMMENTS	(same as in BASIC)

And special statements for the assembler run

The SYMBOL and INSTRUCTION TABLES are really the key to the assembler's success. The SYMBOL TABLE is built from the source code and the INSTRUCTION TABLE. The first pass decodes each statement, and based on the known length of the machine language instructions being generated, creates entries in the symbol table with the symbol's name, and its calculated address where it will appear in the object code. The second pass of the source code by the assembler takes any references to symbols (as a label, or in the parameters) and inserts the corresponding value into the actual object code generated.

Symbol tables can be printed after the assembly has been done, and are very useful in debugging a program, which can be done from the source code, or from the object code using a monitor (another special purpose program).

Here's a simple example:

```
100 .OPT P,OO
110 * = $C000
120 START   =   *
130 LDA #00
140 STA POINTER
150 RTS
160 POINTER =   $00FB
170 .END
```

This would produce a symbol table with 2 entries:

```
START     $C000
POINTER   $00FB
```

and would produce the object code as a string of 1's and 0's, which in hex notation would be:

B50095FB60

Related to addresses, it looks like :

```
$C000  B500   (LDA #00)
$C002  95FB   (STA $00FB)
$C004  60     (RTS)
```

Running the program would load hex zero into location $00FB and do a subroutine return.

The object code would be inserted directly into memory starting at location $C000. The .OPT P,OO is a special control statement that does not get translated into the object code. Ditto with START = *, POINTER = $00FB, and the statements starting with ; (comments only).

Pass one would have generated the SYMBOL TABLE. You would not normally see anything happening at this time. Pass two would display a listing of the assembled code, and the object code.

The listing (or display on the screen) usually contains the object code address for each of the source code statements, along with the source code printed right beside its object code values.

Each assembler will have different control statements and assembly options, but they will usually all conform to the same standards for the format of the actual source code statements. At some point, you can optionally print the SYMBOL TABLE.

When we are developing programs we will be working with the source code and testing the object code, but after development is completed only the object code is used. You will, of course, keep the source code on file (a diskette, maybe) so that future changes can be made to the program just as easily as you created it in the first place.

The object code can be optionally stored on disk, and to run this machine language program, you would load it, like any BASIC program, and execute it. The only difference is instead of 'RUN' you would enter the 'SYS' command followed by the program start address in decimal:

SYS 828

. . . tells the computer to begin executing a machine code program that starts at location 828 decimal. Some Assemblers have the facility to combine BASIC and ASSEMBLER programs in the same source code file. Very handy! The SYS command could be included with the object code so that 'RUN' would execute the 'SYS' which would execute the machine code.

I hope this somewhat general article has helped you to understand how an assembler works. If you are interested in doing machine language programming, then buy an Assembler. . . you won't be sorry!!

Your BASIC Monitor

Bob Drake
Brantford, Ont.

This first of 3 parts describes a monitor program, written in BASIC, that performs most of the functions of a machine language monitor such as TINYMON. The second part will implement a disassembler, and the third an assembler. The programs are written for the Commodore 64. Modifications for the VIC–20 are given. Changes for other computers should be relatively minor as odd programming structures are documented.

Why A Monitor In BASIC?

Many of the students I teach have problems with the concepts of a low level language. What's a low level language you ask? BASIC is considered a 'high level' language because it is very close to ordinary English. Words like PRINT, TAB, FOR, NEXT are nice, simple, easy to understand words. Most high level languages (COBOL, LOGO, FORTRAN) share this property. Some, like APL, don't. A high level language does a lot of things for you automatically. PRINT "HI THERE" has the computer figure out what the word PRINT means, what the quotes mean, set up a loop and print the characters in order in the next locations on the screen.

Low level languages (usually referred to as machine language (ML) or assembler) bear little resemblance to any living language. If we work with numbers (A9, FF, 4C), in base 16 (hexadecimal) we call that machine language or ML. If we use 'assembler' then we get to work with neat words like LDA, STA, DEC, ROL, ROR which are called mnemonics (noo–mon–iks (Greek for memory aid)).A program that lets us write machine language in mnemonics is also called an assembler.

I also find there are problems with the concepts and use of PEEK and POKE. String handling is not much fun either. The capper on all this is a monitor written in machine language. There are monitor programs available for most computers but the average student has an aversion to them. This is a program they can't see (because it isn't written in a language they can read, let alone see), don't understand and which crashes for almost no reason at all. The BASIC Monitor was written to try to solve some of these problems. The program makes heavy use of some simple string techniques and peeks and pokes. A little simple disk and tape filing is thrown in for good luck and completeness.

The program has several commands which correspond to the commands as implemented on most monitors for Commodore computers. All are accessed by single key presses. The menu display is one I have used before to keep the program commands on the screen. Pressing or holding 'RETURN' will erase the screen and bring back the menu.

Displaying Memory

Pressing 'M' for 'Memory' displays the contents of the computer's RAM. This is a first look at the inside of a program and machine language. I usually start teaching this with something like:

```
for i=2048 to 3000 : print peek(i), : next i
```

and we get a list of numbers. We're 'peeking' or looking into those memory locations to see what's there. My favorite spots to look begin (on the C64) at 1024 for the screen, 2048 for BASIC, and 58543 for the power up message. We also type a lot entering that line again and again and again and. . . so it makes sense to create a small program:

```
10 input "from";f
20 input "to    ";t
30 for i = f to t
40 print peek(i),
50 next i
```

and to clean up the presentation

```
10 input "from";f
20 input "to    ";t
30 for i = f to t step 8
35 for s = 0 to 7
40 print peek(i + s);
45 next s
46 print
50 next i
```

Now we get a more or less 'normal' display. Not a terribly neat display since the numbers may be one, two or three digits long, but 'normal'. Normal means this is what you usually get with a Commodore monitor program, eight bytes or characters.

I also like to show my students what those numbers mean. So we add a line like:

```
44 print chr$(peek(i+s));
```

and better to control the printout,

```
44 print chr$(34) chr$(peek(i+s)) chr$(34);
```

CHR$(34) is the quote (") and keeps the display under control when you hit a clear screen command or some other screen command.

If you've followed me to here and tried this out you probably have one awful screen display. Nasty things happen too. Colours change; the screen clears; you get up and down cursor movements. All sorts of horrible things happen. I've even broken into the ML monitor on my PET doing this. VERY nasty.

HEXADECIMAL

Let's clean up that display. First, we'll fix the numbers. As humans, we use decimal arithmetic. That means we count from 0 to 9 and then start reusing those symbols to form 10, 11 and so on. Probably we use decimal because we have ten fingers. Most microcomputers use hexadecimal arithmetic. They count from 0 to 15. (I don't know what that says about the number of fingers they have.) Now counting from 1 to 10 is easy. How the heck do we count from 0 to 15? You may think all you do is count from 0 to 15 (I mean didn't I just do that?) but the rules of numbering say you can only have one symbol for each number in the base set. For example zero is 0, one is 1 and so on but 10 requires a zero and a one. We're fine till we hit 10 through 15 and then we start reusing our symbols. Well, to count rom 0 to 15 we need 16 symbols. Zero through 9 are already okay. Now we only need 6 more. The symbols usually used are A, B, C, D, E, and F. A is 10, B is 11 and so on. F is 15.

The method of converting a number from 17 to 255 from decimal to hex is simple. Divide by 16. The quotient is the first digit, the remainder is the second. So. . . 163 becomes:

```
163/16 = 10 = $A
```

with a remainder of 3.

Thus 163 = $A3. Usually a dollar sign is written in front of hex numbers. So 163 = $A3. Sometimes you will see hex numbers written with an H preceding. So 163 = $A3 = hA3. To do the conversion from decimal to hex on the computer, I use a little subroutine.

```
110 h$="0123456789abcdef"
111 by%=by/16              :rem quotient without decimal
112 r=by-by%*16            :rem remainder
113 f$=mid$(h$,by%+1,1)    :rem first hex digit
114 s$=mid$(h$,r+1,1)      :rem second digit
115 by$=f$+s$              :rem 2 digit hex number
```

Lines 111 to 115 can be condensed to:

```
7020 by$=mid$(h$,by/16+1,1)
     +mid$(h$,by-int(by/16)*16+1,1)
```

H$ contains all the symbols for hex. The correct first digit is found with the MID$(H$, BY% + 1,1). BY% includes an INT function to get rid of the decimal after dividing. You need the +1 because hex numbers start at zero and H$ doesn't have a zeroeth character but does have a first character. We get the second character the same way using the remainder.

These subroutines can be called directly from the monitor by pressing C for Calculator. Enter a number and it is converted to hex. If the number starts with either $ or h, then the hex value it represents is converted to decimal.

Memory Display. . . Again

To fix up the string display, we have to remember that certain peeked numbers will cause trouble. They are:

```
 13 = $0D is a carriage return
 20 = $14 is the delete character
 34 = $22 is a quote
141 = $8D is a shifted carriage return
```

You can take care of the peeked values like this.

```
1150 t$=""
1160 for s = 0 to 7
1170 by=peek(m + s)
1210 if by=13 or by=20 or by=34 or by=141 then by=32
     : rem convert to spaces
1220 t$=t$+chr$(by)
1230 next s
1240 print q$ t$ q$: rem q$ is a quote
```

T$ is a string of eight characters we build out of the peeks. It is "nulled" or emptied in line 1150. BY is the value peeked.

Notice that 1210 changes the illegal characters to spaces and 1220 appends it to T$. Line 1240 prints the string between quotes so that all the characters of T$ print.

Registers

The registers are memory locations in the CPU or central processing unit that the computer uses for addition, counting, and transferring data. They are referred to on the 6500 series of chips as the accumulator (AC), the x register (XR), the y register (YR) and the status register (SR). Commodore makes it easy to look at these in the C64 and VIC 20. Copies of their values are stored in locations 780 to 783. Just peek them, convert to hexadecimal and print.

Poking Memory

Poking memory is sort of a first step to writing machine language. Poking is the reverse of peeking. Instead of looking at a spot in RAM and seeing what numbers are there, you decide what numbers you want put into RAM and poke puts them there. Those numbers can represent instructions (like PRINT, only simpler) or data (like "HI THERE"). At this level of the program you have to know the numbers and what each does. You could for example say:

Put a 5 into the accumulator	$A9 05	(169 5)
Add 2	$69 02	(105 2)
Stop – return to BASIC	$60	(96)

To do this poke 169,5,105,2,96 into 5 memory locations and then run that program. The BASIC Monitor allows you to poke the values in hex, the way they are normally listed and run the program from the monitor. Just press P for POKE, pick a spot to put your program ($33C is pretty good) and enter bytes $A9 05 69 02 60 as your program and fill the rest of the line with 00 (stop). You can run the program by pressing G for GO.

Within the program, we can re–use our memory display to see what is already in memory. When we poke values we need them in decimal. The references you find for machine language are in hex. We must convert hex to decimal. All we do is take the first digit, multiply by 16 and add the second digit. This subroutine is a little fancier than that. It accepts hex numbers 1, 2, 3, or 4 digits long. To make it work, I force the hex number to be 4 digits long by adding "0000" (leading zeros) then taking the four right hand characters. Line 7440 looks at the number the computer assigns to the first letter of the hex number. Line 7450 is the fancy line. It converts this ASCII value to a number from 0 to 15 and multiplies by the appropriate power of 16. The +7*((m1>64)) is zero if m1>64 is false (i.e. m1 represents a number 0 to 9) and minus one if m1 represents a letter (A to F).

```
7410 m$=right$("0000"+m$,4)
7420 m=0
7430 for i=1 to 4
7440 m1=asc(mid$(m$,i,1))
7450 m=m+(m1-48+7*((m1>64)))*16↑(4-i)
7460 next i
7470 return
```

SAVING And LOADING

The subroutines at 4000 and 4500 save a range of peeked values as a tape or disk file. Disk errors are checked in lines 4800–4870.

GO

RUNning a machine language program from BASIC is easy. Just type SYS (the same way you would type RUN) and the address (location in memory) where your machine language program starts. SYS is the ML equivalent of RUN, except RUN will start with the first line of BASIC unless you specify another (eg. RUN 100) – SYS will not default to an arbitrary location, it **must** have an address specified. That's all the subroutine at line 5000 does.

PRINTER

I wanted printer dumps of the memory peeks and registers. The subroutine at 6000 and the PRINT#PR, in the routine at line 1000 accomplish this in a simple manner. It's not terribly fast, but it works easily.

That's pretty well it. You might try running this little program as well as the one given above. By the way, be sure to check the registers in the one above to make sure you get seven. Starting at $33C, poke values of 20 44 E5 60. You should clear the screen on your C64. (Use 20 5F E5 60 on your VIC 20). Press RETURN to get back to the menu.

The program listing follows. The command menu shows selections that aren't available yet. Don't let that throw you. In the next part of this article, we'll add the Disassembler and after that the Assembler. Use the line numbers as shown. If you don't, the additions and changes to come won't fall into place. See you then!

```
100 rem basic monitor* copyright 1984 * r.drake*
    free to copy-not to sell
101 rem * this program listing includes both the
     commodore 64 and vic 20
102 rem * versions of the basic monitor program
103 rem * the overall program is that for the c64.
104 rem * changes for the vic are carefully notated within
    the listing.
105 rem to enter the c64 version, just type in the listing ignoring
106 rem * lines referenced by vic*
107 rem * to enter the vic version, enter the replacement
    lines as noted.
108 rem * line numbers on the first lines of changes
    are important,
109 rem * second lines are not. do not enter lines 101-109
110 h$ = "0123456789abcdef"
120 q$ = chr$(34) : cr$ = chr$(13)
130 open 1,3 : rem screen
140 open 2,4 : rem printer
150 p = 1 :rem printer off
160 print " S ";
170 print chr$(142) " rbas-mon R : rm R emory
    rd R is ra R ssemble rg R o r* R printer"
180 print " rc R alculator rp R oke rr R egisters
    ex R it rs R : rl R data"
181 rem vic*replace 170,180 with 182,183
182 print chr$(142) " rm R em rd R is ra R smbl
    rg R o r* R prntr";
183 print " rc R alc rp R oke rr R eg ex R it
    rs R : rl R ";
190 for i = 1 to 40 :rem vic*use 22 for 40
200 print " ' "; :rem shift-*
210 next i
220 r$ = " xmrpslg*c "
230 r = len(r$)
240 get a$
250 if a$ = " S " or a$ = cr$ then goto 160
260 if a$ = mid$(r$,r,1) then goto 300
270 r = r-1
280 if r = 0 then goto 230
290 goto 260
300 if r<>1 then 350
310 close 1
320 close 2
330 print " rend "
340 end
350 on r gosub 0,1000,2000,3000,4000,
    4140,5000,6000,7500
360 fl = 0
370 goto 230
1000 rem memory
1010 print " rdisplay memory "
1020 print " hold rshift R to pause: rreturn R to stop "
1021 rem vic*replace line 1020 with line 1022
1022 print " hold rshift R to pause    rreturn R to stop "
1030 gosub 4280
1040 if t<=f then t = f + 7
1050 if f<0 or t<0 or f>65535 or t>65535 then 1260
1060 for m = f to t step 8
1070 for pr = 1 to p
1080 n = m
1090 gosub 7030:rem convert to hex
1100 print#pr, by$ " : ";
1110 t$ = " "
1120 for s = 0 to 7
1130 by = peek(m + s)
1140 gosub 7000:rem 2 digit hex
1150 print#pr, by$ " ";
1151 rem vic*replace 1150 with 1152,1153
1152 if s/2 = int(s/2)then print " r ";:if pr = 2 then print#pr," ";
1153 print#pr, by$; " R ";
1160 if by = 13 or by = 141 or by = 20 or by = 34 then by = 32
1170 t$ = t$ + chr$(by)
1171 if s<>7 then t$ = t$ + " ":rem vic*add this line
1180 next s
1190 print#pr, q$ t$ q$:rem vic*print#pr, "    " q$ t$ q$ :rem 4
     spaces
1200 next pr
1210 if peek(653) then 1210    :rem look for shift key
1220 get a$ :if a$<>cr$ then 1250 :rem look for return key
1230 m = t
1240 goto 1250
1250 next m
1255 if m>t then get a$ : if a$<" " then 1255
1260 return
2000 rem registers
2010 print " rdisplay registers "
2020 for pr = 1 to p
2030 for m = 0 to 3
2040 by = peek(m + 780): rem not available on pet
2050 gosub 7000
2060 print#pr,mid$( " ac:xr:yr:sr: ",m*3 + 1,3)by$,
2070 next m
2080 print#pr
2090 next pr
2100 return
3000 rem poke memory
3010 print " rpoke memory "
3020 j = 0
3030 print
3040 input " from " ;m$
3050 gosub 7120
3060 f = m
3070 t = f + 7
3080 print
3090 gosub 1060
3100 for k = 0 to 7
3110 input " QQ byte " ;m$ :rem vic*use 3 up cursors
3120 print tab(5 + k*2 + k ); " r " m$; " R " ;
3121 rem vic*replace 3120 with 3122, 3123
3122 if k/2<>int(k/2)then print " r " ;
3123 print tab(5 + k*2);m$;
3130 print :rem vic*print chr$(13);
3140 gosub 7120
3150 poke f + k,m
3160 next k
3170 input " more  y[3 lefts] " ;a$
3180 if a$<> " y " then return
3190 f = f + 8
3200 print " QQ "
3210 goto 3070
4000 rem save
4010 print " rsave "
4020 s$ = " w "
4030 gosub 4280 : if f = -1 then goto 4130
4040 gosub 4380 : if fi$ = " " then goto 4130
```

```
4050 open3,dv,2,fi$
4060 gosub 4460
4070 print#3,f;cr$;t;cr$;
4080 for i=f to t
4090 print#3,peek(i) cr$;
4100 next i
4110 fl=1
4120 gosub 4460
4130 return
4140 rem load
4150 print" r load"
4160 s$="r"
4170 gosub 4380
4180 open3,dv,2,fi$
4190 gosub 4460
4200 input#3,f,t
4210 for i=f to t
4220 input#3,a
4230 poke i,a
4240 next i
4250 fl=1
4260 gosub 4460
4270 return
4280 rem from-to
4290 input"from";m$
4300 gosub 7120
4310 f=m
4320 print" Q",,
4330 input"to";m$
4340 gosub 7120
4350 t=m
4360 if f<0 or t<0 or f>65535 or t>65535 then
       print" r values out of range":f=-1
4370 return
4380 rem file name & device
4390 dv$="" : input" r R ape or r d R isk";dv$
4400 fi$="" : dv=1-7*(dv$="d") : if dv$="" then 4420
4410 input"file name";fi$
4420 if fi$="" then print"aborted" : return
4430 if len(fi$)>16 then fi$=left$(fi$,16)
4440 if dv=8 then fi$="0:"+fi$+",s,"+s$
4450 return
4460 rem disk status
4470 if fl=0 then open 4,8,15
4480 input#4,b$,b$
4490 print"disk status: " b$
4500 if fl=0 then return
4510 close3
4520 close4
4530 return
4540 stop
5000 rem go
5010 print" r run ml program"
5020 input"address";m$
5030 gosub 7120
5040 input"are you sure";a$
5050 if a$="y" then sys m
5060 print" r done
5070 return
6000 rem printer
6010 if p=1 then 6050
6020 p=1
6030 print" r printer off"
6040 goto 6070
6050 p=2
6060 print" r printer on"
6070 return
7000 rem 2 digit dec to hex
7010 by$=mid$(h$,by/16+1,1)
       +mid$(h$,by-int(by/16)*16+1,1)
7020 return
7030 rem 4 digit dec to hex
7040 by$=""
7050 n=n/4096
7060 for i=1 to 4
7070 n%=n
7080 by$=by$+chr$(n%+55+7*(n%<10))
7090 n=(n-n%)*16
7100 next i
7110 return
7120 rem hex to dec
7130 m$=right$("0000"+m$,4)
7140 m=0
7150 for i=1 to 4
7160 m1=asc(mid$(m$,i,1))
7170 m=m+(m1-48+7*((m1>64)))*16^(4-i)
7180 next i
7190 return
7500 rem calculator
7510 print
7520 input"number";n$
7530 if left$(n$,1)="$" or left$(n$,1)="h" then 7580
7540 n=val(n$)
7550 gosub 7030
7560 print" Q ",,"$" by$:rem vic* print lab(7) "$" by$
7570 goto 7610
7580 m$=mid$(n$,2)
7590 gosub 7120
7600 print" Q ",,m :rem vic* print tab(8) m
7610 return
```

Finding PI Experimentally

Michael Bertrand
Madison, WI

Michael Bertrand holds an MA in Mathematics from the University of Wisconsin. Here he presents a new approach to that age old puzzler.

Computers have opened up the possibility of doing experimental mathematics. Suppose, for example, we wish to know the area of a circle of radius one. Fit the circle snugly inside a square of side two (see figure 1). Paste this picture to the wall and randomly throw darts at it, making sure that they always fall inside the square (but not necessarily the circle — a few darts will land in the corners outside the circle). We expect the proportion of darts landing inside the circle to equal approximately the ratio of the area of the circle to the area of the square. That is:

$$\frac{\#\text{ darts falling inside circle}}{\text{total }\#\text{ of darts thrown}} = \frac{\text{area of circle}}{\text{area of square}}$$

Now the area of the square = $2^2 = 4$, so:

$$\text{area of circle} = 4 * \frac{\#\text{ of darts falling inside circle}}{\text{total }\#\text{ of darts thrown}}$$

Two French mathematicians, the Comte de Buffon (1707–1788) and Pierre Laplace (1749–1827), were the first to discuss problems like this. Intrepid experimenters took up the challenge over the years with mixed results. With a computer, however, the physical experiment can be simulated with random numbers. And unless you can throw 90 darts per second, the computer does it faster.

Now the area of a circle of radius one is pi*1^2 = pi — in fact this could serve as the definition of pi. Thus finding the area of a circle is identical to finding the value of pi.

Programming this experiment is facilitated by concentrating on the upper right hand quarter of figure 1 — see figure 2. The analysis for figure 2 is similar to that above, since we have scaled down both circle and square by a factor of 4. Thus:

$$\begin{aligned}\text{pi} &= 4 * (\text{area of quarter circle}) \\ &= 4 * \frac{\#\text{ of darts falling in qtr circle}}{\text{total }\#\text{ of darts thrown}}\end{aligned}$$

In our computer experiment the "darts" are of course just random numbers. If s = an initial seed value between 0 and 1, and n = the number of trials, then the following BASIC program performs the experiment n times:

```
100 for i=1 to n : gosub200
110 if u↑2 +v↑2 < 1 then c=c+1
120 nexti:end
200 gosub300 : u=s : gosub300 : v=s
210 return
300 s=197*s : s=s-int(s)
310 return
```

Subroutine 300–310 is a random number generator — I don't trust Commodore's. Keep in mind that a point (u,v) in the unit square is within the quarter circle if and only if $u^2 + v^2 < 1$ – this is line 110 – and the "hits" are counted by variable C.

This works fine, but is slow — about 7 trials per second. I rewrote the program in machine language on the Commodore 8032, improving run time by a factor of 13 (about 90 trials per second). My strategy must be identical to BASIC's, since the machine language gives the same result as the above BASIC program every time. I depend heavily on the floating point accumulator ROM routines for numerical operations, as BASIC does. The actual value of pi, correct to 7 decimal places, is 3.141592. Here are the results of the program for a seed value of s = .49032371:

1,000,000	approximation of pi	variance from actual value
1,000	3.128	-.01359 1 sec
10,000	3.1332	-.00839
100,000	3.13572	-.00587
1,000,000	3.140852	-.00074 3 hours, 5 min

If we knew the radius of the earth and the last value here were used for pi in the formula:

$$\text{circumference} = 2*\text{pi}*\text{radius},$$

then the value given for the earth's circumference would be off by less than 6 miles.

Running experiments is not a very efficient way of calculating pi compared to evaluating power series expansions. My purpose is rather to present an understandable example of computer simulation, or what is aptly called the "Monte Carlo" method. The technique is of great value when alter-

native analytical methods are unknown or prohibitively difficult.

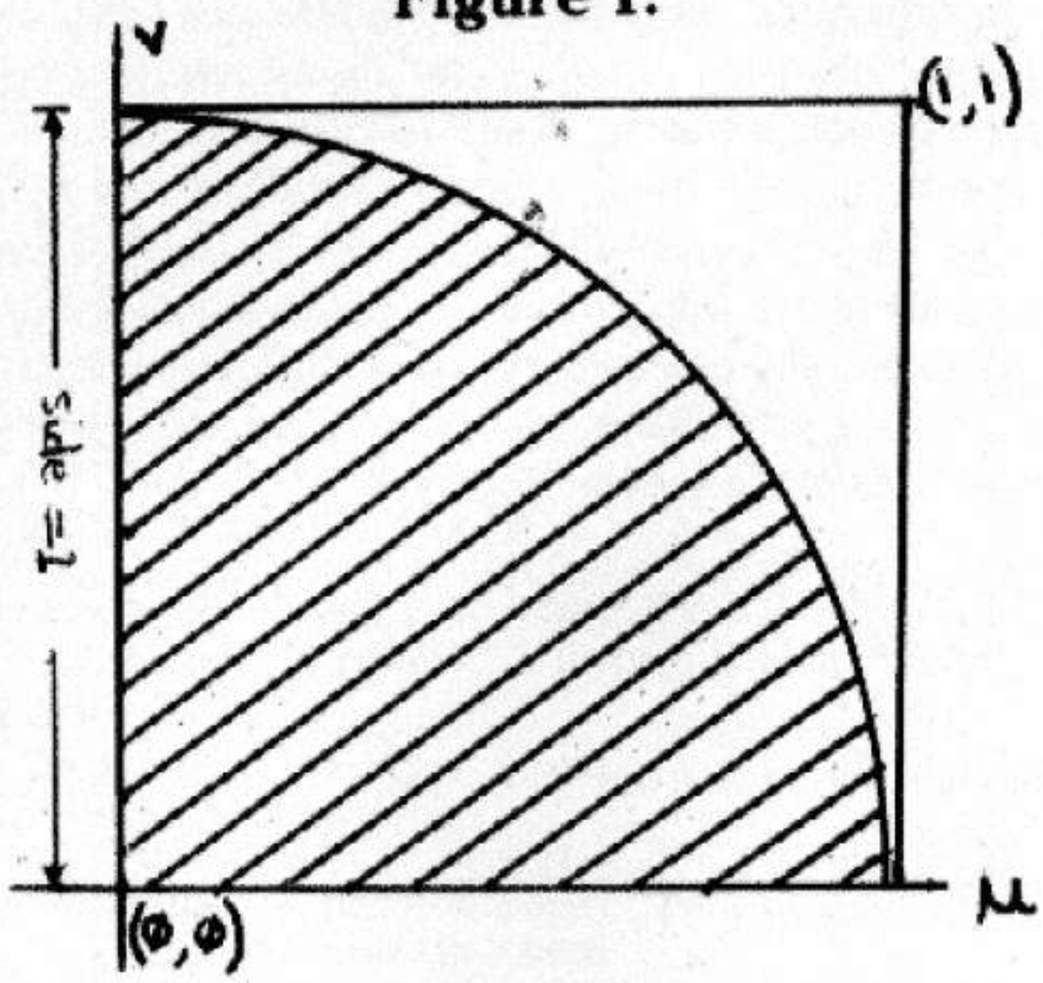
Following is a short annotation of the ml program:

$033c–0340: floating point work area
0341–0345: temporary storage of (random value)^2
0346–034a: 197 in floating point format: (136,69,0,0,0) decimal
034b–034f: 1 in floating point format: (129,0,0,0,0) decimal
0350–0352: total # of trials (n) in 3–byte format — ie, base 256
0353–0355: # of "hits" in 3–byte format — ie, base 256
0356–037b: subroutine to put new random value in $033c and (random value)^2 in fpacc#1
037c–03d6: main routine to do comparison n times and count "hits"

The bit of code at $03cd–03d3 allows interruption of ml execution by pressing the up arrow ($5e ASCII). I find this invaluable in debugging ml programs. It doesn't hurt in the final version either: additional run time is minimal, and program execution can be resumed from BASIC with "goto180" since all necessary values are saved.

This program shows how to put BASIC's floating point ROM routines to work without BASIC's overhead. Run time is generally improved by factors of 12 to 15. If this is what you need for number–crunching applications, then check out those floating point ROMs.

Figure 1.



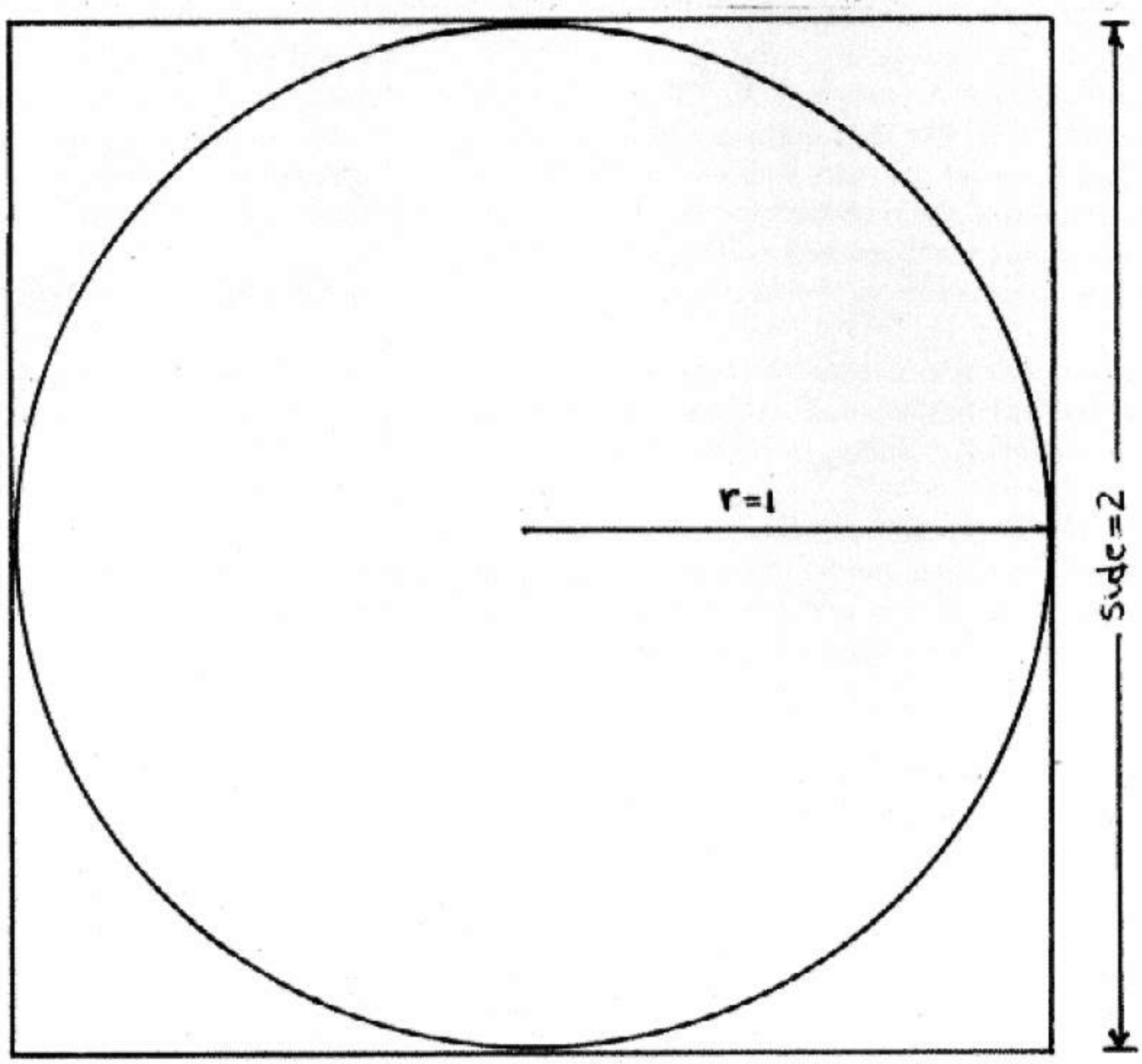


Figure 2.

```
10 rem ************************************************
15 rem *                                              *
20 rem *      * pi program — calculates pi experimentally.      *
25 rem *        machine language depends heavily on the         *
30 rem *        floating point accumulator rom routines.        *
35 rem *                    BASIC 4.0 ONLY                      *
40 rem *      * press the up arrow to interrupt ml execution -  *
45 rem *        execution can be continued with 'goto180'.      *
50 rem *                                                        *
55 rem *        * data/prg at $033c-$03d6 (828-982 dec)         *
60 rem *                                                        *
65 rem *        + + + +   by michael bertrand   + + + +         *
70 rem *                                                        *
75 rem ************************************************
80 :
90 print "■";
100 input " number of trials";n : print
102 if n>16777215 then print " # of trials must be < 16777216" :print:goto100
110 input " seed (between 0 and 1)";s : print
112 if s<=0 or s>=1 then print " seed must be between 0 and 1" :print:goto110
120 m=828:gosub200          :rem * puts seed, in fp format, into (828,829,. . .)
130 s=197:m=838:gosub200 :rem * puts 197, in fp format, into (838, 839,. . .)
140 s=1:m=843:gosub200     :rem * puts 1, in fp format, into (843, 844,. . .)
144 n1=n-1:d3=int(n1/65536):n1=n1-d3*65536:d2=int(n1/256):d1=n1-d2*256
148 poke 848, d3 : poke 849, d2 : poke 850, d1
170 for i=851 to 982 : read x : ch=ch+x
175 poke i, x : next : if ch<>13424 then print "data error" : end
180 sys 892
184 c=65536*peek(851) + 256*peek(852) + peek(853)
188 print " approximation of pi=" 4*c/n : print : end
192 :
194 rem * subroutine 200-230 puts real number s,
196 rem * in floating point format, into memory
198 rem * locations [m,m+1,m+2,m+3,m+4]
200 e=int(log(s)/log(2)) : p(0)=129+e
210 p=(s/2↑e-1)*128:p(1)=int(p):r=p-p(1)
220 for i=2 to 4 : p=r*256 : p(i)=int(p) : r=p-p(i) : next i
230 for i=0 to 4 : poke m+i, p(i) : next i : return
232 :
300 data   0,   0,   0, 169,  60, 160,   3,  32, 216, 204, 169
310 data  70, 160,   3,  32,  94, 203,  32,  66, 205,  32,   2
320 data 206,  32, 137, 201, 162,  60, 160,   3,  32,  10, 205
330 data 169,  60, 160,   3,  32,  94, 203,  96,  32,  86,   3
340 data 162,  65, 160,   3,  32,  10, 205,  32,  86,   3, 169
350 data  65, 160,   3,  32, 157, 201, 169,  75, 160,   3,  32
360 data 145, 205, 201, 255, 208,  19, 238,  85,   3, 173,  85
370 data   3, 208,  11, 238,  84,   3, 173,  84,   3, 208,   3
380 data 238,  83,   3, 173,  82,   3,  72, 206,  82,   3, 104
390 data 208,  21, 173,  81,   3,  72, 206,  81,   3, 104, 208
400 data  11, 173,  80,   3,  72, 206,  80,   3, 104, 208,   1
410 data  96,  32, 228, 255, 201,  94, 240, 248,  76, 124,   3
```

Translating a BASIC Program Into Machine Language

Chris Zamara
Downsview, Ont.

Did you ever write a nifty and clever game in BASIC, only to discover that once you added all the features you wanted, the game was no fun to play because it was too slow? Did you then vow that 'someday' you'd write that program in machine language, putting it off indefinitely, because, while understanding the basics of machine language, you didn't know where to start? Well, if you are in a similar predicament with any sort of BASIC program, fear not: it can sometimes be very easy to translate a program into machine language, or at least use machine language subroutines to speed things up. Read on and see how the humble little BASIC program in Listing 1 went on to become the machine language program in Listing 3. While you're at it, try typing the programs in on your Commodore 64. They graphically illustrate some physical laws of motion, and can be fun to play around with.

Using the BASIC interpreter built into the C64 is an excellent way to develop a program. Programs are easy to enter, modify, and de-bug. Many programs can be used in their final form as BASIC code, but for real-time simulations or similar applications, BASIC just isn't fast enough. When this is the case, we have to go to machine language, or at least use one or more machine language subroutines with the BASIC code.

Even if you plan on using machine language right from the beginning, it is usually convenient to use BASIC to get the logic of the program working right, and then 'translate' the BASIC program into machine language. Many people are familiar with the basic concepts of machine language, but fear the jump from the world of BASIC, intimidated by machine language's lack of floating-point variables, multiply and divide functions, etc. Fortunately, you can usually get around the above limitations, without using routines from ROM, and without writing lengthy subroutines. The kind of program that requires the speed of machine language is usually a simple computer model, or simulation, of some kind (such as a game). Such a program lends itself well to machine language, and can usually be written using simple integer operations, or using table-lookups for complex functions.

Rocket Simulator Example Conversion

To illustrate conversion of a BASIC program to machine language, I will use the relatively short program "ROCKET" in Listing 1. This program simulates the acceleration of a rocket (represented by a sprite) under the influence of its engines (pushing it up), and gravity (pulling it down). When the program is running, pressing the space bar turns the rocket's engines on, and releasing it cuts them. The rocket accelerates as it would under these conditions, neglecting wind resistance. The rocket's thrust and the strength of gravity are variable. Further, this program has a few extras, allowing you to try to land the craft without crashing (the minimum crash velocity is also adjustable). To change any of the three parameters displayed at the top of the screen while the program is running, press BREAK, change the values to whatever you want, then press RETURN. Note that at faster rocket speeds, the sprite's movement is somewhat jerky. This is because the sprite is moving more than one line at a time, to make it move faster. To make the sprite move just as quickly, but only one line at a time for smooth motion, we have to go to machine language.

Before we start translating the program to machine language, an explanation of how it works: First the program loads sprite definitions for the ship, two types of engine flames, and the 'crash' shape into sprite pages 200-203 (you can define your own sprites, or type in the program in listing

2 to create the sprite definition file. If you are the impatient type, delete line 190 and don't bother with the sprite definition for now: your 'rocket' will look funny, but the program will still work). Then some sprite parameters are POKEd into the VIC II video chip. The main program loop begins at line 410. The variable 'VELOCITY' is the speed of the rocket, and actually indicates how many Y values will be skipped on each movement of the sprite. Each time through the loop, 'GRAVITY' is added to this variable, and if the space bar is pressed, 'THRUST' (the strength of the rocket's engines) is subtracted from it. GRAVITY AND THRUST remain constant unless manually changed by first stopping the program. The rocket's 'flames' are also turned on and flickered if the space bar is pressed: the flame sprite is enabled, and the two flame shapes alternately selected. The variable 'VELOCITY' is then added to the Y position of the sprite, and 'Y' is stored in the sprite's Y register. The flame's Y register is also updated (20 units below the rocket). Additional logic stops the ship at the top and bottom of the screen, and indicates a crash or good landing. These extra features will not be incorporated in our machine language translation, but left as an exercise for the ambitious reader.

Translating the first part of the program, setting up the sprite parameters (lines 210–280 in Listing 1) is no problem. POKEs are replaced by LDAs and STAs. This is done in lines 320–440 in the assembler program of Listing 3. I decided to add sound effects for the blasting of the rocket's engine in the machine language version, so a few SID chip parameters are set up in lines 460–520. So far, so good, but now comes the hard part. How do we deal with the variables?

The main variables used to simulate the rocket's motion are THRUST, GRAVITY, VELOCITY, and Y. In the BASIC program, these variables are floating point, and assume fractional values. How do we handle this in machine language? Well, the easiest thing to do is to use large integers for all calculations (we can use more than one byte for a single variable), and then divide Y down by a large amount to get the sprite's Y value, which must be between 0 and 255. Did I say divide? In machine language? Egad! Actually, we can divide by powers of 2 very easily, just by shifting a byte (using LSR or ROR) right one or more times. But if we are using two bytes to store a value (two bytes is a word), it's even easier if we want to divide by 256. All we have to do is use the most significant byte. For example, if 'Y' (called 'ROCKY' in the machine language program) is stored as a word, the first byte in the word is the least significant byte, and the next byte is the most significant byte. The most significant byte increases by 1 when the least significant byte generates a carry, i.e. it gets bigger than 255. If we use the most significant byte as the Y value for the sprite, we have effectively divided the variable 'ROCKY' by 256. This is the way I wrote the original version of the program, using a word to store the abovementioned variables.

The variables increased so quickly, though, that a very long delay was needed each time through the loop to keep the rocket's speed reasonable. The sprite's Y value ended up being increased by more than one unit at a time, so I decided to use three bytes per variable. In lines 210–240 of Listing 3, you can see three bytes allocated to the important variables. Now, when we use the most significant byte of 'ROCKY', we are actually dividing the variable by 65,536! This gives us very good accuracy, and even at very high rocket velocities, the sprite only moves one line at a time.

What about negative values? We certainly need negative values in this program, since the velocity is positive when the rocket is going down, and negative when rising. Luckily for us, the 6502 deals with numbers represented in two's complement form, so that we can represent negative values. In two's complement, a negative value always has its most significant bit (MSB) set. Using a single byte variable as an example, negative 1 is represented by 255, negative 2 by 254, etc. Inverting all bits in a byte and adding 1 will make a negative number positive, or a positive number negative. In this way, a byte can hold values from –128 to +127. Two bytes can store values from –32768 to +32767, and three bytes, what we are using in this program, can store values from around minus to plus 8.4 million. Thus, no special code has to be written to deal with negative values; we can simply add and subtract numbers freely, regardless of their sign.

To add or subtract these three byte variables, we have to add one byte at a time. For example, see lines 640–740 in Listing 3 where gravity is added to velocity. When an addition is performed in the 6502 via the ADC instruction, the value of the carry flag is always added to the result (ADC is an acronym for ADd with Carry). The reason for this will become apparent. Before adding the least significant bytes of the variables together, we clear the carry flag with the CLC instruction. The addition is then performed, in this case storing the result back into 'VELOCITY'. If the result of this addition causes a carry past the most significant bit, the carry flag will be set. Thus, when we add the next most significant byte of the variables (without clearing the carry flag first), the carry will be added in. Likewise for the final, most significant byte. We could add together any number of bytes in this manner, representing huge integer values, or very accurate floating point values.

Now that the basics of how to handle the main variables have been determined, let's translate the actual code. Keep in mind that this will be a very 'loose' translation: the

sequence of some events may be mixed up, some features will be added, others dropped. Added will be the rocket engine sound effects. Dropped will be Y value range checking, so that in the machine language version, the rocket will come up through the bottom of the screen after it exits the top, and vice versa.

The main loop in the BASIC program (Listing 1) is in lines 410–520. This translates to the main loop in the assembler listing (Listing 3), lines 580–1290. The BASIC statement, VELOCITY=VELOCITY+GRAVITY translates to the code in lines 640–740 of Listing 3. In line 760, the keyboard is checked, and if the F1 key is pressed, the program returns to BASIC with an RTS. This is so that thrust and gravity values can be changed if desired. If the space bar is not pressed, the thrust portion of code is skipped. This is equivalent to line 440 in the BASIC listing,

IF PEEK(KEYBD)<>SPACE THEN 480.

In the BASIC program, three things happen in the thrust portion of code, which is executed if the space bar is pressed. First, thrust is subtracted from velocity, then the flames is flickered by switching it's shape, and finally, the flame sprite is enabled in line 470. Moving to the assembler listing, the flame is turned on and flickered in lines 810–860. The sound for the flame is also turned on by gating voice 1 in lines 870 and 880. Finally, thrust is subtracted from velocity in lines 900–1000. Note that when subtracting multi-byte variables, the carry flag is first set with the SEC instruction. If the space bar is not pressed, the flame is turned off, the sound disabled, and some time wasted so that the loop takes the same amount of time whether the space bar is pressed or not. This latter feature, implemented in lines 1091 and 1100, is to prevent the rocket from going slower than it should when the space bar is pressed.

Next in the BASIC program, velocity is added to Y. This is done in lines 1150–1240 of the assembler listing.

The final code in the BASIC program loop is the Y value range checking: the rocket is stopped at the top of the screen, and if it hits the bottom, either a crash or a good landing is indicated. These features are not implemented in the machine language version, so that you can add them yourself as an exercise, if you like.

The last thing in the machine language version's main loop is a delay. Even though the variable 'ROCKY' has to reach 65,536 before the sprite moves just one line, the rocket moves too quickly, even with very low values of gravity and thrust! This gives an idea of how much speed we gain by going to machine language. The delay, in lines 1260–1280, wastes about 1100 cycles, or about 1 ms.

Well, there you have the translated program, written in assembler. If you wish to see the program in action, you can type in the source and assemble it, or if you don't have an assembler you can enter and RUN the BASIC loader program in Listing 4. The new machine language version is slightly different from the BASIC one, but it handles the rocket's acceleration in the same way.

What about changing gravity and thrust? Well, if you recall, pressing the F1 key causes a return to BASIC from the program. We might as well use BASIC to accept new values, POKE them into the correct addresses, and re-execute the machine language program. Since speed isn't critical when it comes to entering the values, it's better to use BASIC, since we can more easily change the prompts, and we don't have to call INPUT routines from machine language. The short BASIC program in listing 5, when run with the machine language code in memory (at $C000), will do the trick quite nicely. To change thrust and gravity parameters while you're flying the rocket, just press F1, and reply to the prompts that appear on the screen. If you later want to change values again, the original values remain on the screen to be modified, kept, or changed entirely.

A few additional tips for converting more complex programs, while still keeping the machine language fairly simple:

1) If you need to multiply a variable by a constant, write a specific purpose multiply routine, eg. a routine that multiplies a given variable by 40. A multiply by 40 could be accomplished quickly by shifting the variable left 5 times (with ASL or ROL) to multiply it by 32, then storing that value and shifting the original variable left 3 times to multiply it by 8. Adding these two results will give the number multiplied by 40.

2) Complex functions, such as SIN, COS, etc. can often be looked up in a table. A table of 256 SIN values ranging from −128 to +127 could be stored in memory beforehand by a BASIC program, and looked up as needed. This technique can also be used for multiplication or division by a constant, where the range of multipliers is known, and reasonably small.

Now that you have a few tricks under your belt, go to it! Convert that nifty game to machine language and watch it fly. You might also want to use some of the code from 'ROCKET" as the basis for another program: maybe make the ship rotate, thrust in the X direction, fire, add some landscape on the bottom of the screen. . .

Listing 1

```
100 goto 190
110 ********************************
120 *              "ROCKET"          *
130 *     simulates a rocket under   *
140 *  influence of thrust and gravity *
150 *   use space bar to thrust; break *
160 *  program to change parameters  *
170 *                                 *
180 ********************************
185 rem * delete line 190 if using
186 rem * cassette.
190 fl=not fl : if fl then load "rocket.sprt",8,1
    :rem * load sprite shape definitions *
200 :
210 rem* sprite/variable initialization
220 vic=13*4096
230 poke vic+21, 1 : rem * enable sprite 0
240 poke 2040, 200 : rem * rocket shape
250 poke vic, 150 : rem * x coordinate
260 poke vic+39, 1 : poke vic+40, 1
270 poke vic+2, 150 : poke vic+3, 255
280 poke vic+41,7 : poke vic+42,8
290 keybd=197 : space=60
300 bottm=229 : rem * bottom of screen
310 gravity=.5 : thrust=1.4 : crash=5
320 :
330 rem * take-off initialization *
340 y=bottm : velocity=0
350 poke vic+1,y
360 print "■gravity=";gravity" : thrust=";thrust"
    : crash=";crash" :     cont"
370 get g$ : if g$<>"  "then 370
380 poke 2040, 200
400 :
410 : rem ** main loop **
420 velocity=velocity+gravity
430 poke vic+21, 1 : rem * turn off flame
440 if peek(keybd)<>space then 480
450 : velocity=velocity-thrust
460 : fl=not fl : poke 2041, 202-fl
470 : poke vic+21, 3 : rem * turn on flame
480 y=y+velocity
490 if y>bottm then y=bottm : if velocity>0 then 560
500 if y<50 then y=50 : velocity=0
510 poke vic+1, y : poke vic+3, y+20
520 goto420
540 :
550 rem * crash or good landing *
560 poke 198, 0 : rem * clear kbd buffer *
570 poke vic+1, bottm
580 if velocity>crash then poke2040, 201 : goto340
    : rem * draw explosion *
590 print "■■■■***good landing !***"
600 for delay=1 to 800 : next delay
610 print "■" : poke 198, 0 : goto 340
```

Listing 2

```
100 rem ** create sprites for "rocket"
110 rem ** if using a cassette instead
120 rem ** of disk drive:
130 rem  goto 230
140 rem * also, if using cassette, this
150 rem * program must be run before
160 rem * executing the main rocket program
170 open1,8,12,"0:rocket.sprt,p,w"
180 print#1, chr$(0)chr$(50);
190 for i=0 to 1 step 0
200 read a : if a>=0 then print#1, chr$(a); : next i
210 close1 : end
220 :
230 rem * cassette version starts here *
240 for i=12800 to 13056
250 read a : poke i, a : next i : end
1000 data   0,  16,   0,   0,  56,   0
1010 data   0,  56,   0,   0, 124,   0
1020 data   0, 124,   0,   0, 124,   0
1030 data   0,  68,   0,   0, 116,   0
1040 data   0, 108,   0,   0, 108,   0
1050 data   0, 108,   0,   0, 124,   0
1060 data   0, 124,   0,   0, 254,   0
1070 data   1, 255,   0,   1, 255,   0
1080 data   1, 255,   0,   1, 215,   0
1090 data   1, 187,   0,   1,   1,   0
1100 data   1,   1,   0,   0,   0,   0
1110 data   0,   0,   0,   0,   0,  10
1120 data   4,   0,   0,   8,   0,   8
1130 data   0,  16,   8,  48,   8,   8
1140 data  96,   4,  72, 192,   2,   0
1150 data   2,   1,   0,   4,   0,   1
1160 data 241,  24,   3, 216,  12,   6
1170 data 248,   0, 247,   0,   0, 246
1180 data   0, 112,  59,   0,   0,  63
1190 data 128,   0, 127, 192,  16, 251
1200 data 198,  49, 241, 199,  17, 225
1210 data 238,   0,   0, 124,   0,   1
1220 data 255,   0,   3, 215, 128,   3
1230 data 187, 192,   3, 147, 128,   1
1240 data 147, 128,   1, 211,  64,   1
1250 data 199,   0,   2, 199,   0,   0
1260 data 199,   0,   2,  71,  64,   0
1270 data 110,   0,   0, 109,   0,   0
1280 data 109,   0,   0,  41,   0,   0
1290 data  56,   0,   0,  56,   0,   0
1300 data  16,   0,   0,  16,   0,   0
1310 data  16,   0,   0,   0,   0,   0
1320 data   0,  56,   0,   0, 254,   0
1330 data   1, 215,   0,   1, 199, 128
1340 data   1, 211,   0,   0, 199,   0
1350 data   0, 206,   0,   0, 238,   0
1360 data   0, 120,   0,   0, 112,   0
1370 data   0, 172,   0,   1, 132,   0
1380 data   1,  36,   0,   1,  96,   0
1390 data   0, 144,   0,   0,  16,   0
1400 data   0,   0,   0,   0,   0,   0
1410 data   0,   0,   0,   0,   0,   0
1420 data   0,   0,   0,   0,  64,   0
1430 data  -1
```

Listing 3

```
100 sys700 : rem written on pal 64
110 ;
120 ;  "ROCKET"
130 ;simulates a rocket under the
140 ;influence of thrust and gravity.
150 ;press space to thrust, f1 to
160 ;change thrust and gravity.
170 ;
180 .opt n,oo
190 * = $c000
200 jmp start
210 thrust .byte 14,0,0
220 gravity .byte 5,0,0
230 velocity .byte 0,0,0
240 rocky .byte 0,0,100
250 flame .byte 202 ;flame shape pointr
260 sprty  = $d001
270 vic    = $d000
280 sid    = $d400
290 sound = $d412
300 keybd = 197
310 start  = *
320 ;sprite set-up stuff
330 lda  #1
340 sta  vic + 21 : sta vic + 39
350 lda  #7 : sta vic + 40
360 lda  #200 : sta 2040
370 lda  #150
380 sta  vic : sta vic + 2
390 lda  #100 : sta vic + 1
400 lda  #120 : sta vic + 3
410 lda  #0
420 sta  vic + 16 : sta vic + 23
430 sta  vic + 29 : sta vic + 28
440 sta  53281 : sta 53280
450 ;
460 ;sound set-up stuff
470 lda  #15 : sta sid + 24
480 lda  #0 : sta sid + 14
490 lda  #4 : sta sid + 15
500 lda  #9*16 + 10 : sta sid + 19 ;a/d
510 lda  #11*16 + 8 : sta sid + 20 ;s/r
520 lda  #128 : sta sid + 18 ;ungate
530 ;
540 ;
550 lda  #0 : sta velocity
560 sta  velocity + 1 : sta velocity + 2
570 ;
580 loop = *
590 lda  rocky + 2 ;use msd of ship's y
600 sta  sprty  ;pos'n as sprite y coord
610 clc : adc #20
620 sta  sprty + 2 ;flame's y co-ord.
630 ;
640 ;add gravity to velocity
650 clc
660 lda  velocity
670 adc  gravity
680 sta  velocity
690 lda  velocity + 1
700 adc  gravity + 1
710 sta  velocity + 1
720 lda  velocity + 2
730 adc  gravity + 2
740 sta  velocity + 2
750 ;
760 lda  keybd
770 cmp #4  ;check for f1 key
780 beq  exit
790 cmp #60 ;check for space
800 bne  nospace
810 lda  #3
820 sta  vic + 21 ;turn on "flame"
830 lda  flame
840 eor  #1
850 sta  flame ;make flame flicker
860 sta  2041
870 lda  #129
880 sta  sound ;turn on sound
890 ;
900 ;subtract thrust from velocity
910 sec
920 lda  velocity
930 sbc  thrust
940 sta  velocity
950 lda  velocity + 1
960 sbc  thrust + 1
970 sta  velocity + 1
980 lda  velocity + 2
990 sbc  thrust + 2
1000 sta  velocity + 2
1010 ;
1020 jmp  spcdone
1030 nospace = *
1040 lda  #1
1050 sta  vic + 21 ;turn off flame
1060 lda  #128
1070 sta  sound ;turn off sound
1080 ;waste time to equalize loop time
1090 ;whether space pressed or not
1091 ldy  #10
1100 waste dey : bne waste
1110 ;
1120 spcdone = *
1130 ;
1140 ;add velocity to y position
1150 clc
1160 lda  rocky
1170 adc  velocity
1180 sta  rocky
1190 lda  rocky + 1
1200 adc  velocity + 1
1210 sta  rocky + 1
1220 lda  rocky + 2
1230 adc  velocity + 2
1240 sta  rocky + 2
1250 ;
1260 ;delay (loop 100 times)
1270 ldy  #100
1280 td dey : nop : nop : nop : bne td
1290 jmp  loop
1300 ;
1310 exit rts
1320 .end
```

Listing 4

```
100 rem * data loader for "ROCKET" *
110 :
120 cs = 0        : rem * checksum
130 os = 49152 : rem * object start
140 :
150 read b : if b<0 then 180
160 cs = cs + b
170 poke os, b : os = os + 1 : goto150
180 :
190 if cs<>31815 then print " * checksum error * "
200 :
240 end
1000 data  76,  16, 192,  14,   0,   0
1010 data   5,   0,   0,   0,   0,   0
1020 data   0,   0, 100, 202, 169,   1
1030 data 141,  21, 208, 141,  39, 208
1040 data 169,   7, 141,  40, 208, 169
1050 data 200, 141, 248,   7, 169, 150
1060 data 141,   0, 208, 141,   2, 208
1070 data 169, 100, 141,   1, 208, 169
1080 data 120, 141,   3, 208, 169,   0
1090 data 141,  16, 208, 141,  23, 208
1100 data 141,  29, 208, 141,  28, 208
1110 data 141,  33, 208, 141,  32, 208
1120 data 169,  15, 141,  24, 212, 169
1130 data   0, 141,  14, 212, 169,   4
1140 data 141,  15, 212, 169, 154, 141
1150 data  19, 212, 169, 184, 141,  20
1160 data 212, 169, 128, 141,  18, 212
1170 data 169,   0, 141,   9, 192, 141
1180 data  10, 192, 141,  11, 192, 173
1190 data  14, 192, 141,   1, 208,  24
1200 data 105,  20, 141,   3, 208,  24
1210 data 173,   9, 192, 109,   6, 192
1220 data 141,   9, 192, 173,  10, 192
1230 data 109,   7, 192, 141,  10, 192
1240 data 173,  11, 192, 109,   8, 192
1250 data 141,  11, 192, 165, 197, 201
1260 data   4, 240, 110, 201,  60, 208
1270 data  52, 169,   3, 141,  21, 208
1280 data 173,  15, 192,  73,   1, 141
1290 data  15, 192, 141, 249,   7, 169
1300 data 129, 141,  18, 212,  56, 173
1310 data   9, 192, 237,   3, 192, 141
1320 data   9, 192, 173,  10, 192, 237
1330 data   4, 192, 141,  10, 192, 173
1340 data  11, 192, 237,   5, 192, 141
1350 data  11, 192,  76, 230, 192, 169
1360 data   1, 141,  21, 208, 169, 128
1370 data 141,  18, 212, 160,  10, 136
1380 data 208, 253,  24, 173,  12, 192
1390 data 109,   9, 192, 141,  12, 192
1400 data 173,  13, 192, 109,  10, 192
1410 data 141,  13, 192, 173,  14, 192
1420 data 109,  11, 192, 141,  14, 192
1430 data 160, 100, 136, 234, 234, 234
1440 data 208, 250,  76, 113, 192,  96
1450 data  -1
```

Listing 5

```
100 rem * basic code for rocket program
110 rem * machine language rocket routine
120 rem * starts at $c000 (49152)
130 :
140 p = 49152 : print " S "
141 rem * for cassette use, delete
142 rem * line 145 and first run the
143 rem * sprite create program
145 if peek(12801)<>16 then load " rocket.sprt " ,8,1
150 print "  press space to thrust, "
160 print " press f1 to change thrust and gravity "
170 sys(p)
180 poke 198, 0 : rem * clear kbd buffer *
190 input "  thrust " ;th
200 input " gravity " ;gr
210 poke p + 3, th and 255 : poke p + 4, th/256
220 poke p + 6, gr and 255 : poke p + 7, gr/256
230 goto 150
```

A Few Of The Stranger 6502 Op Codes Explained

Richard Evers

For a large percentage of assembly language programmers, most of the 6502 instruction set is never actually used. Instructions like LDA, STA, INC, DEC, BEQ, BNE, JMP, JSR are usually all that are required for most applications. What I want to do today is go beyond these instructions and advance into the lesser known, or lesser understood op codes to try to shed some light on their workings.

The Carry Flag

Let's start off a little slow. The term CARRY is used quite often within most machine code programs. ADC – add with carry, SBC – subtract with carry, SEC – set the carry flag, CLC – clear the carry flag, BCS – branch on carry flag set, and BCC – branch on carry flag clear, are all of the instructions that fit into this category.

Carry is a flag that is set to one (on) when an arithmetic operation takes place in which the result generated goes beyond the 8 bit maximum limit. Before an addition, carry has to be cleared with a CLC, so we can later tell if we have exceeded the 8 bit maximum. When the carry flag has been set, the op code BCS – branch on carry set, will always succeed. On the other hand, the instruction BCC, branch on carry clear, will always succeed if the arithmetic operation has not gone past the 8 bit limit and carry is clear.

Before a subtraction operation is performed, the carry flag has to be set. This is due to the fact that if your resulting value is less than zero, which is beyond the 8 bit realm, the carry flag will be cleared, and a test can be made of this condition with the BCC – branch on carry clear, instruction.

Branching

A little note to slip in before we go much further. Confusion seems to run rampant about how a branch actually works. Below is a very useless program to demonstrate how a branch instruction calculates the address to go to when you branch forward or backwards.

```
027a  here = *
;
027a  b8     clv
027b  50 03 bvc everywhere  ;branch forward to
                            'everywhere'
027d  there = *
;
027d  b8     clv
027e  70 fa  bvs here       ;branch backwards to 'here'
;
0280  everywhere = *
;
0280  b8     clv
0281  50 fa  bvc there      ;branch backwards to here
```

As you can see from the program above, a branch forward uses the next address after the branch line as location $00, and increments the location from there until the branch destination has been reached. A branch backwards works the same only in the opposite direction. The branch offset itself is considered as location $FF, and is decremented from that point backwards until the destination is reached. The maximum branch in any direction is $80 characters, or a full page between the two directions. Beyond this you must use a JMP instruction.

The Overflow Flag

Our next condition to cover is the overflow flag. In this category are BVS - branch on overflow flag set, BVC - branch on overflow flag clear, and CLV - clear overflow flag. The overflow flag will be set when an arithmetic calculation exceeds 7 bits in magnitude.

During signed bit arithmetic operations within the 6502, positive numbers are stored in true binary, with negative numbers stored in twoś compliment binary. Two's compliment means that all the bits but bit zero are reversed. A value of +7 = %0000 0111 where a value of -7 = %1111 1001. The high bit is used to signify the sign of the value. When an addition operation exceeds +127, the high bit will be set, thus making it now appear to be a negative number. If tested for, overflow would flag that a sign correction routine should be called. Essentially, the overflow flag works just like the carry flag except carry is set when a calculation exceeds 255, overflow is set when a calculation exceeds 127.

The overflow flag can also be set externally from a pin on the microprocessor chip. It's not an interrupt so you must anticipate activity on this pin, therefore it's rarely used this way. But it's very fast, much faster than an interrupt where you must save your registers before actually servicing the interrupt. Commodore disk drives use this pin to signal 'data ready' from the read/write head.

The BIT instruction is another one that alters the OVERFLOW flag. BIT will be covered later in this article.

The Zero Flag

BEQ – branch if equal to zero (result true), and BNE – branch if not equal to zero (result false). What do these mean to you? These instructions use the Z (zero) flag to test if the last operation performed was equal to zero or not. If you were to load the accumulator with 255, then compare it to 40, they would not be equal to each other. In this operation, the Z flag would be set to zero to signify that the test failed, or the result was false. The instruction BEQ would fail where BNE would succeed.

Consider a simple loop routine as shown below. CPX (or CPY if you had used the Y register) is not required to test for 'Not Equal'. BEQ and BNE test the result of the last operation performed.

```
        ldx #0
loop    inc $8000, x
        inx
        bne loop
```

No actual comparison was made, but the branch will succeed until the X register returns to a zero value. As I mentioned earlier, BEQ and BNE test the results of the last operation performed. This can sometimes give rather unexpected results, so the Compare op codes do have purpose too.

The Negative Flag

Two more branches for your contemplation. BPL – branch if plus (negative flag set to zero), and BMI – branch on minus (negative flag is set to one). Both of these are very handy at times when a test to determine the actual values of characters encountered is required. Take for example the little bit of code below:

```
loop  jsr $ffe4   ;get a character from the keyboard
      beq loop    ;if no key pressed, go back to try again
      sec
      sbc #48     ;ascii zero
      bmi loop    ;was below an ascii one
      sbc #3
      bpl loop    ;was above an ascii three
* followed by your code that uses the value between 1 and 3 *
```

In this example, the test does a very good job of allowing you to pick and choose precisely what want from the keyboard.

Decimal Mode

Enough of the branches, lets get into further arithmetic instructions. SED – set the decimal flag, and CLD – clear the decimal flag, refer to a mode that will allow you to utilize BCD, binary coded decimal form of arithmetic calculations. In this mode, each nybble of a byte holds a single decimal number, that is 0 through 9. A full byte can have a value up to 99 decimal. ADC, SBC, BCC and BCS all work in the same way with this mode of operation, with carry being set if decimal 99 is exceeded. As before, carry has to be set before subtraction, and cleared before addition.

This mode of operation is not one in which Commodore have made great strides in using to any advantage. When your machine is first powered up, a CLD instruction is executed to ensure that all arithmetic calculations are performed in regular binary. On entrance into the machine language monitor, the decimal flag is also set to zero, as an added incentive to avoid the BCD blues. As it stands, not a single BASIC routine takes advantage of BCD. Perhaps in a future version of CBM BASIC, BCD will finally be used.

Stack Operations

The next instructions just crying for a little explanation are : PHA – push the accumulator onto the stack, PLA – pull the accumulator from the stack, PHP – push the processor status onto the stack, PLP – pull the processor status from the stack, TSX – transfer the stack pointer to the X register and TXS – transfer the X register to the stack pointer. Except for TSX and TXS, these instructions are used to either get characters from the stack, or store characters on the stack. Quite often, before execution of a machine language program, the current state of the computer should be saved for future use. Below is a quick routine to demonstrate a method to save everything for future retrieval:

```
php  ;push the processor status onto the stack
pha  ;push the contents of the accumulator onto the stack
tya  ;transfer contents of y reg into the accumulator
pha
txa  ;transfer contents of x reg into the accumulator
pha
tsx  ;transfer the stack point to the x register
txa  ;transfer contents of x reg into the accumulator
pha
```

As you probably know, the stack is a 256 byte area in RAM held at $0100 hex to $01ff. The trick is that it is used for much more than most can comprehend. That small block of RAM seems to be the centre of attraction for almost all of the computers functions. Not only is the stack responsible for holding return addresses for various functions, but it is also used for most of BASIC's arithmetic calculations.

Further, hundreds of currently available machine language programs make use of the stack in some pretty un-orthodox ways. Some of the most ingenious methods of program protection known to date make use of the stack as a special hideaway spot for tricky bits of code to trap the unwary hacker. Unless you are in the mood for some pretty hairy protection techniques, stay away from this one. For the balance of masochists in our reading audience, plow ahead and create your own versions of the spiral of death, as one particular technique has come to be known. Challenges like this seem to make life more interesting.

Boolean Operations

And now, time for a bit of boolean fun. In our boolean lunchbox we have : AND – and memory with the accumulator, ORA – or memory with the accumulator and EOR – do an exclusive or of memory and the accumulator.

For a lot of the examples in the rest of the article, I will be referring to values in binary format. The reason for this will soon become apparent.

AND

The diagram below shows exactly how this instruction operates.

```
accum.  1001 0110 / 150 decimal
AND     1101 1101 / 221 decimal
        ---------------------
result  1001 0100 / 148 decimal
```

To AND a value with another, you simply match bits up and determine which ones matched. The rule is:

AND: result is true if one AND the other

The bits that were ON in both values remain ON, the balance are set OFF. In this way, if you wanted to keep a certain calculation within your boundaries, you would AND the result with the highest value that you want. The result can be worth less or equal to the ANDed value, but cannot exceed it.

OR

Again, the diagram below will show its operation.

```
accum.  1001 0110 / 150 decimal
OR      1101 1101 / 221 decimal
        ---------------------
result  1101 1111 / 223 decimal
```

When you OR a value with another, every corresponding bit that is turned ON in either value is left ON in the final result. The rule:

OR: result is true if one OR the other

In this way you can stop a value from dropping below what you want. OR the value with the lowest value you want and the result will always be equal or higher.

For all of you that can remember back a few issues, a couple articles appeared that mentioned a technique of OR'ing the characters read from disk files with 64 to help stop the rude temptations of pseudo control characters. If you were to directly read in the contents of a program file, and print those contents to your screen, every now and then a character will sneak in with the sole purpose in life to make your days on earth difficult. Those characters attempt to make you believe that they are control characters, and set out in their task of clearing you screen, setting windows and doing other equally rotten things. A quick OR with 64 will rob these imposters of their power and knock them back down into the ranks of the others. A pretty long winded way to say how not to let the value drop below 64 decimal.

EOR

The next, and last, boolean operator is really a mixture of numerous boolean logic circles, and will take a little bit longer to explain.

As before, lets break it down into binary for the explanation

```
accum.  1001 0110 / 150 decimal
EOR     1101 1101 / 221 decimal
        ---------------------
result  0100 1011 /  75 decimal
```

One step further will show that EOR is incredibly powerful to use

```
result  0100 1011 /  75 decimal (result from above EOR)
EOR     1101 1101 / 221 decimal (previous EOR value)
        ---------------------
accum.  1001 0110 / 150 decimal (original accum. content)
```

As the above diagrams have shown, EOR will perform a bit flip operation. The rule here is:

EOR: result is true if one OR the other, but not both

Each pair of bits that do not match are turned ON, pairs that do match are turned OFF. If a pair contained two 1's or two 0's, then they would be turned OFF. If the pair contained 1 and 0, or 0 and 1, the result would be ON. When an EOR is performed again with the same original value, the bits would be returned to their prior state. The cursor is a good example of EOR in effect.

One particularly useful method of using EOR is in the encryption of data within programs. If you wanted to have a password access system for your program, but didn't want anyone to be able to read the passwords, EOR them with a known variable before storing them away. When you read them back, EOR them with the same value again to un-encrypt them for usage. As long as the EOR variable is kept secret, the passwords are pretty secure. A little imagination at this point will help you design a very difficult encryption system to break. As I said before, EOR is very useful.

Bit Shifters

And now, away from boolean logic and into stranger bit beating for the enthusiastic programmer. Into this category I include ASL - shift memory or accumulator one bit to the left, LSR - shift memory or accumulator one bit to the right, ROL - rotate memory or accumulator and the carry flag one bit left, and ROR - rotate memory or accumulator and the carry flag right one bit. Each a pretty valuable instruction, and each worthy of some explanation.

ASL: Shift memory or accumulator one bit to the left.

accum. 1001 0110 / 150 decimal

After an ASL, the result would be 0010 1100, or 44 decimal with the carry flag set. The carry flag being set represents the result exceeding 255. Therefore the result of this operation = 44 + 256 = 300, or double the original value. Thus we now have a way to multiply by two from within assembly code.

LSR: Shift memory or accumulator one bit to the right.

accum. 1001 0110 / 150 decimal

After a LSR, the result would be 0100 1011, or 75 decimal with the carry flag clear. This allows division by ywo in machine code. If bit zero was originally ON, carry would be set after the operation to signify that a fraction was encountered. Therefore, if the accumulator content was 0110 0101 before the operation (101 decimal), after a LSR the bit structure would be 0011 0010 with carry set, which is equal to 50 plus carry, or 50 with a remainder.

ROL: Rotate memory or accumulator and carry one bit left.

As the name implies, this operation is a rotate, not a shift. ROL rotates the bits around in a circle in a counter clockwise direction.

As usual, lets start with the accumulator holding the value below, but show carry as the left most bit of 9 bits. To start, carry is clear.

```
Carry Accumulator (start)
  0   1001 0110 = 150 decimal with carry clear
```

After a ROL, the result would look like this:

```
Carry Accumulator (result)
  1   0010 1100 =  44 decimal with carry set or 300 decimal
```

As can be seen, ROL is similar to an ASL, but with a difference. The most significant bit becomes the current carry status, and the prior carry status becomes the least significant bit. Used in conjunction with ASL, some really heavy duty multiplication can be preformed. By ROL'ing everytime ASL generates carry, you could

double the original value, then repeatedly double the result each time. By stringing multiple ROL's together, calculations of unbelievable length can be achieved.

ROR: Rotate memory or accumulator and carry one bit right.

By now you probably know what this one can do for you. As ROL was similar to ASL, ROR is similar to LSR with the added advantage of rotating the carry bit along with the rest.

As can be expected, ROR will rotate all the bits in a clockwise direction moving the low bit into carry, and the carry bit into the high bit. A binary value of %0110 0101 with carry clear would become %0011 0010 with carry set after a ROR is executed.

Below is a program to demonstrate how ROR can be used for multiplication of two 8 bit numbers to generate a 16 bit result. Below that is another demonstration program, this time for division of a sixteen bit number by an eight bit number generating an eight bit result. This example uses the ROL instruction to produce its expected result.

```
; *** let's multiply 87 * 16 ***
;
027a  a9  57  lda  #%01010111   ;87 decimal
027c  85  fb  sta  loval
027e  a9  10  lda  #%00010000   ;16 decimal
0280  85  fc  sta  hival
0282  a2  08  ldx  #8
0284  a9  00  lda  #%00000000   ;zero
0286  18      clc
;
0287  loop = *
0287  6a      ror  a
0288  66  fb  ror  loval
028a  90  03  bcc  spot
028c  18      clc
028d  65  fc  adc  hival
;
028f  spot = *
028f  ca      dex
0290  10  f5  bpl  loop
0292  85  fc  sta  hival
0294  60      rts
```

The 16 bit result can be found in locations $fb + $fc (low byte/ high byte)

```
; *** divide 31587 by 227 ***
; 31587 = %0111 1011 0110 0011
;
027a  a9  63  lda  #%01100011   ;low byte of
027c  85  59  sta  loval        ;16 bit value
027e  a9  7b  lda  #%01111011   ;high byte of
0280  85  5a  sta  hival        ;16 bit value
0282  a9  e3  lda  #227         ;divided by
0284  85  5b  sta  by           ;8 bit value
0286  a2  08  ldx  #8
0288  a5  5a  lda  hival
028a  18      clc
;
028b  loop = *
028b  26  59  rol  loval
028d  2a      rol  a
028e  b0  04  bcs  spot
0290  c5  5b  cmpby
0292  90  03  bcc  nextspot
;
0294  spot = *
0294  e5  5b  sbc  by
0296  38      sec
;
0297  nextspot = *
0297  ca      dex
0298  d0  f1  bne  loop
029a  26  59  rol  loval
029c  85  5a  sta  hival
029e  60      rts
```

The 8 bit result can be found in $59 with the remainder in $5a.

Testing Memory and 'Hiding' Code

Our last instruction for today is the BIT instruction. BIT is really rather strange, to say the least, for it does not alter memory in any way, it just changes three of the processor register flags.

The three flags involved are the Z – zero flag, V – overflow, and N – negation. The zero flag is set if the boolean expression accumulator AND memory fails, or the zero flag is clear if accumulator AND memory succeeds. Bit 6 of the memory location of the BIT operation is transferred into the V flag, and bit 7 of the memory location is transferred into the N flag. BIT is a pretty good way to test if bit seven has been set in a specific memory location, or bit six if that is more to your liking. What ever the case, it really does have its good points.

There is one more interesting use for BIT, if you are at all interested in program protection. As Jim Butterfield and a host of thousands have already pointed out, an absolute BIT op code before a LDA instruction helps hide load instructions quite nicely. When disassembled, the code might look like:

```
027a  24  a5  77  bit $77a5
027d  48      pha
027e  20  95  02  jsr $0295
```

But in reality the code was written with this in mind:

```
100  *=$027a
110  .byte $24  ;bit op code
115  lda $77    ;get the low byte of basic from chrget
120  pha
125  jsr $0295  ;and do something else for a while
```

To enter this code, location $027a is bypassed and location $027b is chosen instead. This different location of entry will ensure that the code is used correctly.

There are probably more instructions that you are unfamiliar with but, for today, thats all I intend to cover. If you really don't understand a particular instruction, send us a letter and we'll attempt to answer it in one of our future issues. If we receive mountains of questions, perhaps another article like this one will hit the pages of Transactor again. 'Till later, thanks for allowing me to climb inside your mind and jiggle your bits about.

Getting BASIC To Communicate With Your Machine Code

Darren Spruyt
Gravenhurst, Ont.

There are several ways of communicating to your ML program from BASIC and I wish to outline what I think are the most common, and best-liked. The first method is by the USR command, It allows one to pass a single variable into the floating point accumulator#1. Second is the POKE/SYS method, in which you POKE values into memory, and then your code loads the values back from memory. Thirdly, the SYS/WEDGE method, which allows about 10–15 parameters to be passed by using the CHRGET pointer. And finally, the variable/sys method which allows an infinite (realistically about 5428) amount of parameters to be passed. I have tried to have information for three machines – 64/VIC/PET-CBM – the information for the 64 is directly stated, the PET/CBM info is in square brackets '[ ]', while the VIC info is in backslashes '//'.

The POKE Method

The POKE method is quite simple and straight-forward. There are two options we can use with the POKE format. One involves a memory location (or several) which we write data into from BASIC, and later, read back from our assembly code. The other just involves POKES from BASIC.

The first option involves POKEing a value to a location from BASIC and then retrieving it from ML with a LDA, LDY or LDX command. e.g POKE 8192,100/LDA $2000. Of course the hex address used with the LDA, LDX, or LDY command must match the value given in the POKE command. i.e. $2000 is hex for 8192.

This method is simple and easy to use. It has a few drawbacks: it is code consuming in BASIC if more than one parameter is to be passed; and to transfer large numbers, the number must be broken into smaller byte-sized pieces and is realistically limited to integer values. To transfer a value of 'X' in two byte unsigned arithmetic (range 0–65535) would require two pokes:

```
POKE ADD,INT(X/256):POKE ADD+1,X-INT(X/256)*256
```

where ADD = memory location to POKE to and X is the value to be transferred.

The latter option may be slightly easier to use, but only 3 parameters can be passed with a range of 0–255 and it only works on the C-64. In the 64 there are three locations in page 3. These are locations 780, 781 and 782. The first is for the accumulator, the second for the X-register and the last for the Y-register.

If we POKE a value of 128 in the location for the accumulator, and zero in the others we can demonstrate the effect. Code in memory somewhere a BRK command. Location $4000 is a good place. . . and SYS to it:

```
poke 780, 128 : poke 781, 0 : poke 782, 0
poke 16284, 0 : sys 16384
```

(You will need a machine language monitor program installed first, like Supermon64, to get the proper reaction) If we now look at the register display (the info that comes up when a BRK is executed) we will see that the accumulator (ac) holds a value of $80 (hex for 128) and both X (xr) and Y (yr) will be 0. With this we can see that values POKEd into 780, 781, and 782 appear in the respective registers when control is transferred to an ML program. So we have an easier way to transfer 3 parameters (range of 0–255).

We can also set what the processor status register will be the instant that control is transferred to our program. We do this by POKEing 783 with the correct value to set whatever flags we wish to have set. One note: When our ML program terminates with a RTS, the values that are in the accumulator, X-reg and Y-reg and the status register are then placed back into the same memory locations mentioned above so we can then examine them with a PEEK command.

The USR Method

The USR method uses one of BASIC's lesser known and used commands: USR. An illustration:

A = USR(100)

The value of 100 is placed into the Floating Point Accumulator #1 (FPACC#1–this is a group of 6 bytes in zero page in which BASIC performs all of its arithmetic operations) for your program to work on and when an RTS instruction is encountered from your code, the current value that is in FPACC#1 is then placed in the variable 'A' and control is returned to BASIC.

Unlike SYS, USR has no starting address indicated in its format. However, the starting address in specified in the USR jmp vector: $0310–784 ($0000–0002) [$0000–0002]. The first byte of each is the jump opcode JMP or $4C. The next two bytes are the destination in the standard 6502 format, destination lo and hi respectively. Here is an example of a computed–GOTO routine:

```
$2000 jsr  $b7f7 [$c92d]/$d7f7/ ;convert FPACC#1 to in-
                                 teger
$2003 jsr  $a613 [$b5a3]/$c613/ ;search for line
$2006 bcs  $200b                ;carry set if found
$2008 jmp $a8e3 [$bf00]/$c8e3/ ;undefn'd statement
$200B pla                       ;remove calling address
                                 put on stack
$200C pla                       ;by the 'USR' routine
$200D jmp $a8c5 [$b850]/$c8c5/ ;cont GOTO routine
```

Note: This routine does not return FPACC#1 into the variable because we do not return control to the USR ROM routine, instead we return control to the GOTO routine.

Before this code can be executed, The USR vector must be pointed to it. This entails:

poke 785, 0 : poke 786, 32 (poke 1, 0 : poke 2, 32)
[poke 1, 0 : poke 2, 32]

This is just an idea of what can be done with it, however it can be used to do anything you might require of it eg. data transfers through user port/RS–232 or many other applications.

This last and simple program using the USR function just multiplies the given value by 10.

```
$2000 jsr  $bae2 [$cc18]/$dae2/ ;multiply FPACC#1 by
                                 10
$2003 rts                       ;end of routine
```

We set the USR destination by using the above POKEs, and whenever we call it, it will return a value multiplied by 10. e.g. PRINT USR(115) will result in '1150' being printed.

The Wedge Technique

This technique requires some basic knowledge of the CHRGET routine. This routine @ $0073 on VIC/64 and @ $0070 on PET/CBM is a routine that is called everytime BASIC needs the next character in a program to execute. Inside this routine is a pointer ($7A)–64/VIC and ($77)–PET/CBM that points at all times to the most recent character fetched from BASIC text space.

Consider a program with a line such as '10 SYS 49152:GOTO 1000'. As control is transferred to our ML program, the CHRGET pointer is left pointing to the next character to be taken from BASIC code, in this example the colon ':'. The pointer is not changed while our program is executing, but we can change it by calling some ROM routines, and being able to do this can be very advantageous to us.

Maybe you have seen a program with an SYS like this :'SYS 49152,lo,hi', this program uses the wedge technique for the ML program to obtain the two parameters that follow the SYS.

In the example immediately abovee, the CHRGET pointer will be left pointing to the comma (,). We can test for a comma by calling JSR $AEFF [$BEF5–PET/CBM] /$CEFF–VIC/. If the comma is present, control will return to us and the CHRGET pointer will be increased by 1; otherwise '?SYNTAX ERROR' will result. The next step would be to call the routine evaluate expression at $AD9E [$BD98] /CD9E/, this will leave the type of expression in $0D [$07] /$0D/ . A value of $00 means a numeric result, while a value of $FF means it is alphanumeric or string. The numeric value would be left in FPACC#1, while a string value would occupy three bytes, pointed to by an indirect pointer at ($64) [($61)] and /($64)/. The order of these bytes are: length of the string, the low–byte of the address and the high–byte of the address of the string.

Once we have copied the information that we needed i.e. string length, destination lo and hi, we must then clean the 'descriptor stack.' We do this by the following code: LDA $64/LDY $65/JSR $B6DB. If we do not do this, we may eventually have a 'FORMULA TOO COMPLEX' error. Note:this is only needed when working with strings.

Listing 4 shows us how to retrieve a floating–point variable, where the syntax for calling this routine is 'SYS8192;EXP' where 'EXP' is a numeric expression. Listing 5 is an example of crunching out an integer value from the expression:

syntax for this one is the same as above, but the allowable range is 0–65535. Listing 6 retrieves a single byte value (0–255) and prints it. Listing 7 is a psuedo POKE routine, where syntax is 'SYS8192;ADD,VAL' where ADD is the memory location to be POKEd to, and VAL is the value to be put there (0–255). Finally, Listing 8 will display a string expression using the wedge technique. If we wanted to transfer about six variables, we can alternate between the 'check for comma' and the 'evaluate expression' routines.

The Variable Method

BASIC, in storing variables (simple, not arrays), uses seven bytes for each variable. The first two are the name and the next five are used differently for the different types. As more variables get defined, BASIC keeps adding them to a table that it keeps in memory, the start of this table is pointed to by (\$2D) [(\$2A)] /(\$2D)/ and the end is pointed to by (\$2F) [(\$2C)] /(\$2F)/. Lets examine the first two bytes representing the name.

There are two bytes used, and BASIC only recognizes the first two characters of a variable. Now you say 'How does BASIC then tell a two character string variable from a two character floating–point variable?' e.g. AA\$ from AA. We know that the first character of a variable is a letter and that the second character can be either a letter or number. BASIC stores the variables' names using the PETSCII character value, but with a twist.

In the PETSCII range that the characters occupy, the MSB (Most Significant BIT) is never touched since letters occupy the range of 65–93 (\$41–\$5A; %01000001 – %01011010) and numbers occupy 48–57 (\$30–\$39; %00110000 – %00111001). In both ranges, the MSB (bit 7) is never touched. BASIC uses the MSB, in both character positions, to represent the four types of variables (string, floating–point, integer and functions). Table 1 represents this idea.

Table 1

Type	MSB–1st char	MSB–2nd char
F–P	0	0
INT	1	1
STR	0	1
FNCT	1	0

One character variables have a zero placed in the second position, but the MSB is still manipulated accordingly.

Variable Names plus their storage values (hex) (Excluding functions)

A – \$41 and \$00, AB – \$41 and \$42
C\$ – \$43 and \$80, CZ\$ – \$43 and \$DA
Z% – \$DA and \$80, D1% – \$C4 and \$B1

The rest of the 5 bytes remaining in each variable type are used as follows:

Floating–Point: sign + exponent and four bytes of mantissa.
String: length, string address lo, string address hi, last two not used.
Integer: value hi, value lo, last three not used.

When the last several bytes are not used (string, integer) the not used ones are filled with zeros. Question: Why the unused bytes in the integer and string variables? Answer: to be able to use a constant seven additive to increase speed when searching through the variable tables.

A Real Example

A machine language monitor will be needed, so LOAD it now if you don't already have one in the computer. Type:

```
NEW <return>
10A=100:B$="DARREN":C%=100
RUN
```

Now enter the monitor with SYS 8 [SYS4, PET/CBM]. Type:

```
m 002d 002d <return> [$2A on PET/CBM] /$2D on VIC/
```

The pointer (\$2D) [(\$2A)] /(\$2D)/ tells us where the simple variables list starts and (\$2F) [\$2C] /(\$2F)/ tells us where arrays start and variables end. In our example we should have seen:

```
.:002d 20 08 35 08 35 08 ed 97   —64
.:002a 20 04 35 04 35 04 00 80   —PET/CBM
```

Now type:

```
m 0820 0835 <return>
```

Display of :

```
.:0820 41 00 87 48 00 00 00 42   —PET/CBM is identical,
.:0828 80 06 0f  08 00 00 c3 80     but at $04 20 – 04 35
.:0830 00 64 00 00 00 00 00 00
```

Perhaps it is easier if we break into groups of seven from the beginning.

```
name  s/e   mantissa
.:0820 41 00  87   48 00 00 00     –1st variable defined
name  len lo hi   unused
.:0827 42 80 06  0f 08   00 00     –2nd variable defined
name  hi lo unused
.:082e c3 80 00 64 00 00 00        –3rd variable defined
```

With this information, we can now write Assembly language subroutines to access the BASIC variables. Number 11 in the list of ROM routines, is the routine used to find a BASIC

variable. Listing 1 is an example of an assembly program to retrieve a string variable 'AB$' from memory, while Listing 2 retrieves an integer variable and finally, listing 3 retrieves a floating point variable.

Well this is the end, and while I hope that everything is correct, it may not be so. If I find any problems (heaven forbid), I will try to get them into the Transbloopers section of the next issue. And finally, I hope that I have presented everything clearly so that all may be able to use these new-found procedures.

ROM Routines

In the following list, the values given outright are for the C-64, while the ones in parenthesis '()' are for PET/CBMs (BASIC 4.0) and the ones in square brackets '[ ]' are for the VIC-20.

1. JSR $AEF7 ($BEEF) [$CEF7] - tests next character of BASIC text for a right bracket ')'.
2. JSR $AEFA ($BEF2) [$CEFA] - tests next character of BASIC text for a left bracket '('.
3. JSR $AEFD ($BEF5) [$CEFD] - tests next character of BASIC text for a comma ','.
4. JSR $AEFF ($BEF7) [$CEFF] - tests next character of BASIC text for the indicated character in the accumulator (using PETSCII codes).

All of the above routines leave the next BASIC character in the accumulator and they exit leaving the Y-register at 0 and the X-reg unchanged.

5. JSR $B79E ($C8D1) [$D79E] - gets a one byte value (0-255) from BASIC text (through CHRGET) and return it in the X-reg.
6. JSR $B1BF ($XXXX) [$D1BF] - converts FPACC#1 into a 2-byte signed integer at $64 ($XX) [$64] in the standard 6502 format—low then high values.
7. JSR $B7F7 ($C92D) [$D7F7] - converts FPACC#1 into a 2-byte unsigned integer at $14 ($11) [$14]
8. JSR $B391 ($C4BC) [$D391] - converts the integer in Y (low value) and A (high value) into a floating point value in FPACC#1.
9. JSR $BBD7 ($CD0D) [$DBD7] - packs what is in FPACC #1 into a memory variable at X (low value) and Y (high value). X and Y point to the data of the variable, not the name.
10. JSR $BBA2 ($CCD8) [$DBA2] - unpacks memory variable into FPACC#1 where A is the low byte and Y is the high byte. X and Y point as mention above (#9).
11. JSR $B0E7 ($C187) [$D0E7] - find variable given name in $45 ($42) [$45] first character and $46 ($43) [$46] second character and returns variables location (start of the data, not the name) in Y (high value) and A (low value), also in $47 ($44) [$47] which is to be used an an indirect index. i.e. LDA ($47),Y (LDA ($44),Y) and [LDA ($47),Y]. Indirect address $5F ($5C) [$5F] points to the start of the name of the variable. It must be remembered that the high bits of the name must be modified dependant on the type of variable being searched for. e.g. to find variable 'AB%':LDA #$C1/STA $45/LDA #$C2/STA $46/JSR $B0E7—for the C-64, will leave the above pointers set to the correct addresses for 'AB%'.
12. JSR $AD9E ($BD98) [$CD9E] - Evaluate expression from where the CHRGET pointer was pointed to. Result type is in $0D-$00 means result was numeric and $FF means result was a string. If result was numeric, the value is in FPACC#1, while is a string, $64 ($61) [$64] — as an indirect index — points to a string of three bytes of which the first is taken to be the length of the string and the next two are the low-byte and the high-byte of the location, in memory, of the string.
13. JSR $AD8A ($BD84) [$CD8A] - Evaluate numeric expression. Calls $AD9E, but then tests type of result, and if not numeric then a 'TYPE MISMATCH' error results. Used if a numeric value is wanted.
14. JSR $BDD7 ($CF8D) [$DDD7] - Prints the number that is currently in FPACC#1. To insure it to work (on the C-64), load the Y-reg with $01 before calling (if not done, what is printed is the absolute value).
15. JSR $AD8D ($BD87) [$CD8D] - check input was numeric.
16. JSR $AD8F ($BD89) [$CD8F] - check input was string. Both This one and previous are called after Evaluate Expression (#12) is called.
17. JSR $B6DB ($C811) [$D6DB] - cleans descriptor stack. LDA $64 and LDY $65 before calling (64/VIC) or LDA $61 and LDY $62 (PET/CBM)
18. JSR $B4F4 ($C59E) [$D4F4] - creates room for new string with length in accumulator. Location for string is at ($33)-64/VIC and ($5F)-PET/CBM.

Listings

Before typing in most of the listings, you should have a machine language monitor installed like Supermon-64. All of the listings presented here are for the C-64, with the help of the ROM Routines section above, one should be able to modify them easily to work on the PET/CBM or VIC.

Listing 1: Assemble at $2000

```
$2000 lda  #$41     ;first letter/bit 7=0
$2002 sta  $45
$2004 lda  #$c2     ;second letter/bit 7=1
$2006 sta  $46
$2008 jsr  $b0e7    ;find variable, given name
$200b ldy  #$02
$200d lda  ($5f),y  ;pointer to start of variable/get length
```

```
                    of string
$200f beq $202b     ;length zero-go
$2011 tax           ;
$2012 iny           ;
$2013 lda ($5f),y   ;get address lo
$2015 sta $14       ;save it
$2017 iny           ;
$2018 lda ($5f),y   ;get address hi
$201a sta $15       ;save it
$201c stx $2100     ;save length
$201f ldy #$00      ;zero y reg
$2021 lda ($14),y   ;get a char from string
$2023 jsr $ffd2     ;print it
$2026 iny           ;in
$2027 cpy $2100     ;printed all the chars
$2029 bne $2021     ;no, get some more
$202B rts           ;end of routine
```

Now anytime a call to $2000 (SYS 8192) is executed, it will print the current value of the string 'AB$'

Listing 2: Assemble at $2000

```
$2000 lda #$c1      ;1st char-bit 7=1
$2002 sta $45       ;save value
$2004 lda #$c2      ;2nd char-bit 7=1
$2006 sta $46       ;save value
$2008 jsr $b0e7     ;find variable
$200b ldy #$00      ;.y=0
$200d lda ($47),y   ;get hi-byte
$200f tax           ;into .x
$2010 iny           ;.y=.y+1
$2011 lda ($47),y   ;get lo-byte
$2013 tay           ;into .y
$2014 txa           ;.a=.x
$2015 jsr $b391     ;convert to floating-point variable in
                    fpacc#1
$2018 ldy #$01      ;.y=1
$201a jsr $bdd7     ;print fpacc#1
$201d rts           ;end of routine
```

Now, anytime this routine is called (SYS 8192), it will print the current value of the integer variable 'AB%'. Note: we had to change the integer to floating-point value before we could print it—this is the cause that makes integer variables are slower than floating-point. The BASIC in the 64 cannot handle integer arithmetic, it converts them to F-P's, then does the operation and then converts the result back into an integer.

Listing 3: Assemble at $2000

```
$2000 lda #$41      ;1st char-bit 7=0
$2002 sta $45       ;save value
$2004 lda #$00      ;2nd char-bit 7=0
$2006 sta $46       ;save value
$2008 jsr $b0e7     ;find variable
$200b jsr $bba2     ;transfer from memory to fpacc #1
$200e ldy #$01      ;.y=1
$2010 jsr $bdd7     ;print fpacc #1
$2013 rts           ;end of routine
```

Now, anytime this routine is called (SYS 8192), the floating-point variable 'A' will be printed.

Listing 4: Assemble at $2000:

```
$2000 lda #$3b      ;chr$($3b)=";"
$2002 jsr $aeff     ;test chrget for char in .a
$2005 jsr $ad8a     ;get numeric value into fpacc#1
$2008 ldy #$01      ;.y=1
$200a jsr $bdd7     ;print fpacc #1
$200d rts           ;end of routine
```

Listing 5: As Listing 4, but change as follows:

```
$2008 jsr $b7f7     ;convert to integer
$200b ldx $14       ;.x=value in $14
$200d lda $15       ;.a=value in $15
$200f jsr $bdcd     ;print .a*256+.x
$2012 rts           ;end of routine
```

Listing 6: As Listing 4, but change as follows:

```
$2005 jsr $b79e     ;get 1-byte value
$2008 lda #$00      ;.a=$00
$200a jsr $bdcd     ;print .a*256+.x
$200d rts           ;end of routine
```

Listing 7: As Listing 4, but change as follows:

```
$2008 jsr $b7f7     ;convert to integer
$200b jsr $aefd     ;check for comma
$200e jsr $b79e     ;get 1-byte value
$2011 txa           ;.a=.x
$2012 ldy #$00      ;.y=$00
$2014 sta ($14),y   ;put value into memory
$2016 rts           ;end of routine
```

Listing 8: Assemble at $2000

```
$2000 lda #$3b      ;.a=$3b
$2002 jsr $aeff     ;test chrget for char in .a
$2005 jsr $ad9e     ;evaluate expression
$2008 bit $0d       ;indicator for type
$200a bmi $200f     ;if neg then string
$200c jmp $ad99     ;'type mismatch'
$200f ldy #$00      ;.y=$00
$2011 lda ($64),y   ;get length of string
$2013 beq $202f     ;if length = 0 then go
```

```
$2015 sta  $97      ;save length
$2017 iny           ;.y = .y + 1
$2018 lda  ($64),y  ;get lo of string address
$201a sta  $14      ;$14 = .a
$201c iny           ;.y = .y + 1
$201d lda  ($64),y  ;get hi of string address
$201f  sta  $15     ;$15 = .a
$2021 nop           ;at this point, the string is
$2022 nop           ;at ($14) and its length is in $97
$2023 ldy  #$00     ;.y = $00
$2025 lda  ($14),y  ;get a character of the string
$2027 jsr  $ffd2    ;print character
$202a iny           ;.y = .y + 1
$202b cpy  $97      ;compare .y to length
$202d bne $2025     ;print another character of the string
$202f  rts          ;end of routine
```

The String Insert Program

This program takes three inputs: position at which to start the insertion and two strings, the one to be inserted and the receive the insert. This program is an exercise in retrieving values using the wedge technique and also the variable technique. The program resides at $C000 on a C-64, but with modifications could be made to fit anywhere else. The syntax for using is:

sys 49152:insert position, variable$, string

The result is then left in 'variable$'. e.g:

a$ = "darren" : sys 49152: 3, a$, "ccc"

will result in A$ being "darcccren".

```
$c000 lda  #$3a     ;
$c002 jsr  $aeff    ;test for character in .a
$c005 jsr  $b79e    ;get a single byte value (0-255)
$c008 txa           ;set flags
$c009 bne $c00e     ;if not zero then fine
$c00b jmp $af08     ;'syntax error'
$c00e stx  $15      ;save the value
$c010 jsr  $aefd    ;checjk for comma
$c013 jsr  $ad9e    ;evaluate expression
$c016 bit  $0d      ;test flag for type
$c018 bpl  $c00b    ;if numeric then syntax
$c01a lda  $65      ;get lo of address
$c01c beq  $c00b    ;if zero, then it was not a variable
$c01e lda  $47      ;get lo of address of string
$c020 sta  $3f      ;save it
$c022 lda  $48      ;get hi of address of string
$c024 sta  $40      ;save it
$c026 ldy  #$00     ;.y = 0
$c028 lda  ($64),y  ;get length of string #1(remember that
                    ($64) pointed to a string of three bytes
                    which were taken to be length, string
                    address lo and string address hi)
$c02a beq  $c00b    ;if length is zero then syntax
$c02c cmp$15        ;compare length with insert location
$c02e bcc  $c00b    ;is it less (insertion impossible)
$c030 beq  $c00b    ;is it equal (use concatenation)
$c032 sta  $c200    ;save length
$c035 iny           ;.y = .y + 1
$c036 lda  ($64),y  ;get string #1 address lo
$c038 sta  $c201    ;save it
$c03b iny           ;.y = .y + 1
$c03c lda  ($64),y  ;get string #1 address hi
$c03e sta  $c202    ;save it
$c041 jsr  $aefd    ;check for comma
$c044 jsr  $ad9e    ;evaluate expression
$c047 bit  $0d      ;test type flag
$c049 bpl  $c00b    ;numeric then 'syntax error'
$c04b ldy  #$00     ;.y = 0
$c04d lda  ($64),y  ;get length of string #2
$c04f  beq $c00b    ;if length is zero then syntax
$c051 sta  $c203    ;save length
$c054 iny           ;.y = .y + 1
$c055 lda  ($64),y  ;get string #2 address lo
$c057 sta  $c204    ;save string #2 lo address
$c05a iny           ;.y = .y + 1
$c05b lda  ($64),y  ;get string #2 hi address
$c05d sta  $c205    ;save it
$c060 lda  $64      ;
$c062 ldy  $65      ;
$c064 jsr  $b6db    ;clean des. stac
$c067 clc           ;
$c068 lda  $c200    ;get string #1 length
$c06b adc  $c203    ;add string #2 length
$c06e sta  $14      ;save length
$c070 bcc  $c075    ;under 255 total length
$c072 jmp $b658     ;'string too long'
$c075 jsr  $b4f4    ;make room for string
$c078 ldx  #$05     ;
$c07a lda  $c200,x  ;
$c07d sta  $22,x    ;transfer pointers to zero page
$c07f  dex          ;string #1-len ' $22, pointer ($23)
$c080 bpl  $c07a    ;string #2-len ' $25, pointer ($22)
$c082 ldy  #$00     ;transfer string #1. . ...
$c084 lda  ($23),y  ;
$c086 sta  ($33),y  ;
$c088 iny           ;
$c089 cpy  $15      ;until insert length achieved
$c08b bne $c084     ;
$c08d tya           ;increase ($33) by .y
$c08e clc           ;and leave result in ($41)
$c08f  adc $33      ;
$c091 sta  $41      ;
$c093 lda  $34      ;
$c095 adc  #$00     ;
$c097 sta  $42      ;
```

```
$c099 ldy  #$00     ;transfer string #2
$c09b lda  ($26),y  ;
$c09d sta  ($41),y  ;
$c09f  iny          ;to memory ' ($41)
$c0a0 cpy  $25      ;finished?
$c0a2 bne $c09b     ;no
$c0a4 tya           ;increase ($41) by
$c0a5 clc           ;value in .y/.a
$c0a6 adc  $41      ;and leave
$c0a8 sta  $41      ;
$c0aa bcc  $c0ae    ;result
$c0ac inc  $42      ;in ($41)
$c0ae lda  $41      ;decrease ($41)
$c0b0 sec           ;
$c0b1 sbc  $15      ;by value in $15
$c0b3 bcs  $c0b7    ;
$c0b5 dec  $42      ;
$c0b7 sta  $41      ;
$c0b9 ldy  $15      ;get insert position
$c0bb lda  ($23),y  ;transfer last half
$c0bd sta  ($41),y  ;of string #1
$c0bf  iny          ;
$c0c0 cpy  $22      ;finished?
$c0c2 bne $c0bb     ;no
$c0c4 ldy  #$00     ;.y=0
$c0c6 lda  $14      ;total length of new string
$c0c8 sta  ($3f),y  ;store in in variable
$c0ca iny           ;
$c0cb lda  $33      ;get string address lo
$c0cd sta  ($3f),y  ;store it in variable
$c0cf  iny          ;
$c089 lda  $34      ;get string address hi
$c08b sta  ($3f),y  ;store it in variable
$c08d rts           ;finished!!
```

This last program, by Jim Butterfield, uses the variable techniques. It is a program to return a string of characters input from the disk. It allows inputs including colons, commas, and semi–colons which, if used with an INPUT# statement in BASIC would cause a 'extra ignored' error. OPEN 1,8,2,"NAME" is needed before this code will work and first variable defined must be a string variable.

```
$02c0 ldy  #$02
$02c2 lda  ($2d),y   ;($2d) is start–of–variables (simple)
$02c4 sta  $0089,y   ;this area is free space
$02c7 iny
$02c8 cpy  #$06      ;is it 6?
$02ca bne $02C2      ;no, lets get some more
```

–after this loop has been executed, the string length is in $8b, string address in ($8c) and both $8e and $8f are zero.

```
$02cc lda  #$01      ;
$02ce jsr  $ffc6     ;command file #1 to talk
$02d1 jsr  $ffe4     ;get a char from file
$02d4 cmp#$0d        ;is it a return character?
$02d6 beq $02e7      ;yes!
$02d8 ldy  $8e       ;get index to string, ' start =0
$02da sta  ($8c),y   ;store it in the string
$02dc iny            ;increase index by one
$02dd sty  $8e       ;save new index value
$02dF cpy  $8b       ;compare it to actual length of string
$02e1 beq $02e7      ;they are equal—no room left
$02e3 lda  $90       ;should always be zero
$02e5 beq $02d1      ;an always branch
$02e7 jmp $ffcc      ;untalk file and return to basic
```

BASIC Loader For String Insert

```
1000 rem string insert
1010 for j= 49152 to 49364 : read x
1020 poke j,x : ch = ch + x : next
1030 if ch<> 25843  then print" checksum error "  : end
1040 rem use sys49152:po,targ$,ins$
1050 data 169,  58,  32, 255, 174,  32, 158, 183
1060 data 138, 208,   3,  76,   8, 175, 134,  21
1070 data  32, 253, 174,  32, 158, 173,  36,  13
1080 data  16, 241, 165, 101, 240, 237, 165,  71
1090 data 133,  63, 165,  72, 133,  64, 160,   0
1100 data 177, 100, 240, 223, 197,  21, 144, 219
1110 data 240, 217, 141,   0, 194, 200, 177, 100
1120 data 141,   1, 194, 200, 177, 100, 141,   2
1130 data 194,  32, 253, 174,  32, 158, 173,  36
1140 data  13,  16, 192, 160,   0, 177, 100, 240
1150 data 186, 141,   3, 194, 200, 177, 100, 141
1160 data   4, 194, 200, 177, 100, 141,   5, 194
1170 data 165, 100, 164, 101,  32, 219, 182,  24
1180 data 173,   0, 194, 109,   3, 194, 133,  20
1190 data 144,   3,  76,  88, 182,  32, 244, 180
1200 data 162,   5, 189,   0, 194, 149,  34, 202
1210 data  16, 248, 160,   0, 177,  35, 145,  51
1220 data 200, 196,  21, 208, 247, 152,  24, 101
1230 data  51, 133,  65, 165,  52, 105,   0, 133
1240 data  66, 160,   0, 177,  38, 145,  65, 200
1250 data 196,  37, 208, 247, 152,  24, 101,  65
1260 data 133,  65, 144,   2, 230,  66, 165,  65
1270 data  56, 229,  21, 176,   2, 198,  66, 133
1280 data  65, 164,  21, 177,  35, 145,  65, 200
1290 data 196,  34, 208, 247, 160,   0, 165,  20
1300 data 145,  63, 200, 165,  51, 145,  63, 200
1310 data 165,  52, 145,  63,  96
```

CIA Timers

Elizabeth Deal
Malvern, PA

Timers on the CIA chip are fun to work with. They are versatile little devices. They can measure time intervals of several microseconds up to several minutes. They can time a duration of a signal being in one logic state, they can be used in continuous, recycling mode, or do a quick count and quit. They can be given a new time value at any time, they can be read reliably, the flags they set can be read correctly, and they can ring a bell for attention . . . need I go on? Let your imagination run wild – you can use those clocks for all sorts of terrific experiments.

They aren't even all that difficult to use once you get the drift of the jargon in the book. I find these translations helpful:

CRA/B is control register A or B – that is a panel of switches on top of clock A and clock B. You flip and turn the switches as you wish (position 0 to set the alarm, position 1 to set the clock – this sort of thing).

ICR is interrupt control register – you tell it which (if any) events should call you (alarm); it tells you the status of events (eight bits have come in). It's a neat communication link. It's two things in one place for two sided talk. The only tricky thing here is that once you've asked it a question (has clock B done its's job?) the status, if the answer is yes, vanishes. Which is wonderful in a way, because it gets ready for the next event, but you may need to remember the status if you are looking at more than one thing. I wish VIC chip was built like that.

In any case, I had my share of problems, and this article discusses just one mode of the timers' operation, TIMING fairly LONG EVENTS, by computer standards, that is.

I wanted the clocks to participate in a simple SID orchestra. But I soon discovered that training a clock to be a musician isn't simple. The clock refused to keep the beat (a contradiction in terms?). At first I thought I coded the whole thing wrong, but subsequent snooping into the operation of the timers revealed that there is a problem built into the CIA chip.

According to the chip description in the Programmer's Reference Guide, the timers can be used in, what they call, 'extended mode'. That's fancy talk meaning timing events longer than about 1/15th of a second, precisely my goal. The chip offers two options in how to use the timers: you can set them and read them in a loop waiting for the elapsed time to pass, or you can set them and ask that they call you (interrupt) when the time is up.

The sort of thing I was coding did not need the speed of attention and the accuracy of the interrupt-generating feature. And since it's devilishly complicated to set up a whole alternate interrupt system immune to crashes, I decided to go the easy way. Let the clock run. I can check it about the time I think it might be done. I knew I had lots of time to spare. So I coded something like this:

```
1 set the timers for some duration
2 do other things, then look at the clock –
3 has the clock set its 'all-done' flag?
4 if not, waste more time i.e. goto 3
5 all done.
```

This didn't always work. Things got stuck. Music sounded rather odd, being off-beat. Some notes sounded twice as

long as they should, and, (very infrequently), the loop never ended crashing the computer which just sat there waiting for that flag to be set. Thinking I had a coding error, I revised my approach. But subsequent snooping revealed that while I may well have coding problems, the chip has problems of its own which the attached program demonstrates in vivid colour and noise.

It includes two tests. You can change the timing by changing variable TM in Basic code. Since I am only interested in events exceeding 1/15th second, Basic uses only one byte for time. This value should be greater than zero.

Test 1 puts the timers in the mode I was trying to use – no interrupt request. When you run it you will hear fairly regular clicking noises and the colours should be coordinated. At some point (normally within the first 5–45 seconds) things fall apart. We miss a click and the colors become unsynchronized. The point of this exercise is to show that timing is taking place, that timer B does in fact count down past zero, but that it sometimes forgets to tell the flag register that it's mission is finished. Hence, to use this mode, the only reliable way seems to be to code around the problem – to watch the clock itself and ignore the flag altogether.

Test 2 does a more difficult thing: it tells timer B not only to count but to interrupt the computer when the time is up. This is a vital mode for critical timing operations, hence we have to trust its reliability. I once thought that it, too, misbehaved. But I can't duplicate the results any more (program changes!). So while I can't be 100% sure it never fails, I think I found a musician after all.

If any of you feel like running test 2 for a good, long time, please share your results with all of us.

References

1. Jim Butterfield, Memory Maps and Machine Language lessons without which I wouldn't know where to even begin doing this sort of thing.

2. R West, Programming the PET/CBM which talks of hardware matters in somewhat simpler terms than

3. Commodore64 Programmer's Reference Guide, hardware appendix. Great book. The hardware pages are rough reading for people new to this sort of thing, but all the information is useful if you can somehow absorb their jargon into your own thinking.


```
1000 rem----------------------------------
1010 rem 6526/cia timer-b  liz deal
1020 rem----------------------------------
1050 tm=08 : mc=832 : b0=176
1052 if peek(mc)*peek(mc+1)<>76*75 then
     for j=832 to 1022 : read v : poke j,v : nextj
1060 i$="" : print" [YtL][Saaaaaaaaaaaaaaaaa] ";
1070 print"     <cr;ctrl;wipe+cr> timeout: :intbit "
1080 input"do test 1 2  ";i$
1090 pokeb0,tm
1100 onasc(i$+"0")-48goto1120,1130
1110 end
1120 sysmc : goto1060
1130 sysmc+6 : sysmc+3 : goto1060
1135 rem source code below can be omitted,
1137 rem data from 5000 on must be entered
1140 rem----------------------------------
1150 sys700    ;pal source code
1160 .opt oo
1170 *=$0340   ;save to $3ff
1180 ;
1190 jmp test1
1200 jmp test2
1210 jmp nmisw
1220 here .word mynmi
1230 ;
1240 p =17*40+28     ;for screen
1250 cia2      =$dd00        ;non-kb cia
1260 ta2       =cia2+4       ;timer a
1270 tb2       =cia2+6       ;timer b
1280 cra2      =cia2+$e      ;ctrl reg a
1290 crb2      =cia2+$f      ;ctrl reg b
1300 icr2      =cia2+$d      ;int ctrl+flags
1310 cia1      =$dc00        ;the other cia
1320 prb1      =cia1+1       ;stop key here
1330 col       =$d800
1340 mask1     =%00000010
1350 mask2     =%10000010
1360 ctrlky    =$fb
1370 anynmi    =$318
1380 nornmi    =$fe47
1390 inex      =$febc
1400 val       =$b0          ;goes into tb2
1410 once      =val+1
1420 to        =col+p        ;to always moves
1430 in        =col+p+1      ;in,noise when
1440 noise     =$d418        ;timerb sets flg
1450 ;----------------------------------
1460 ;tb sometimes fails to set a flag
1470 ;show#1 – not asking for interrupt
1480 ;----------------------------------
1490 test1  =*
1500 lda icr2
```

```
1510 sei : lda #%01111111 : jsr setup
1520 lda #mask1
1530 wait =*
1540 jsr teststop : beq quit
1550 ; watch timer b reload
1560 jsr watch
1570 ; check timeout flag,loop if 0
1580 bit icr2 : beq wait
1590 ; flag worked this time
1600 jsr click : beq wait ;always more
1610 ; back to basic
1620 quit cli : rts
1630 ;---------------------------------
1640 ;show#2 – asking for an interrupt
1650 ;seems to work most of the time
1660 ; = = = = = (can't prove non-failure!)
1670 ;---------------------------------
1680 nmisw =*
1690 ; clr flags in hope of surviving
1700 ; what follows
1710 lda icr2 : lda #%01111111 : sta icr2
1720 ; revector nmi stuff to here
1730 ; this can kill you
1740 lda here : ldx here + 1 : jsr setvec
1750 ; now tell the icr & clocks
1760 lda #mask2 : jsr setup : rts
1770 ;
1780 mynmi =*
1790 pha : lda icr2 : and #2 : beq myi2
1800 txa : pha : tya : pha : jsr click : jmp inex
1810 myi2 pla : jmp nornmi
1820 ;
1830 test2 =*
1840 ; nothing better to do loop
1850 jsr watch : jsr teststop : bne test2
1860 ; set things back to normal
1870 lda #0 : sta icr2
1880 lda #<nornmi : ldx #>nornmi
1890 setvec =*
1900 sta anynmi : stx anynmi + 1
1910 rts
1920 ;
1930 teststop =*
1940 ldx prb1 : cpx #ctrlky : rts
1950 ;
1960 watch =*
1970 ; watches timer b reload
1980 ; ignore quick reading results
1990 ldx tb2 : cpx val : bcc watch9
2000 cpx once : beq watch9
2010 inc to          ;displ timeout
2020 watch9 stx once : rts
2030 ;
2040 click =*    ; & displ flag set
2050 inc in : ldx #8 : stx noise
2060 sig1 dex : bne sig1
2070 stx noise : rts  ;z = 1
2080 ;
2090 setup =*
2100 sta icr2 : sta to : sta in : inc in
2110 ; init clocks (val*ta)whatevers
2120 ldx #$ff : stx ta2 : stx ta2 + 1
2130 inx : stx tb2 + 1 : stx once
2140 ldx val : stx tb2
2150 ; force load time,tb counts ta
2160 ; timeouts, cont mode,clocks run
2170 lda #%00010001 : sta cra2
2180 lda #%01010001 : sta crb2
2190 rts
2200 .end
2210 end
5000 data  76,  75,   3,  76, 151,   3,  76, 106
5001 data   3, 129,   3, 173,  13, 221, 120, 169
5002 data 127,  32, 213,   3, 169,   2,  32, 175
5003 data   3, 240,  13,  32, 181,   3,  44,  13
5004 data 221, 240, 243,  32, 198,   3, 240, 238
5005 data  88,  96, 173,  13, 221, 169, 127, 141
5006 data  13, 221, 173,  73,   3, 174,  74,   3
5007 data  32, 168,   3, 169, 130,  32, 213,   3
5008 data  96,  72, 173,  13, 221,  41,   2, 240
5009 data  10, 138,  72, 152,  72,  32, 198,   3
5010 data  76, 188, 254, 104,  76,  71, 254,  32
5011 data 181,   3,  32, 175,   3, 208, 248, 169
5012 data   0, 141,  13, 221, 169,  71, 162, 254
5013 data 141,  24,   3, 142,  25,   3,  96, 174
5014 data   1, 220, 224, 251,  96, 174,   6, 221
5015 data 228, 176, 144,   7, 228, 177, 240,   3
5016 data 238, 196, 218, 134, 177,  96, 238, 197
5017 data 218, 162,   8, 142,  24, 212, 202, 208
5018 data 253, 142,  24, 212,  96, 141,  13, 221
5019 data 141, 196, 218, 141, 197, 218, 238, 197
5020 data 218, 162, 255, 142,   4, 221, 142,   5
5021 data 221, 232, 142,   7, 221, 134, 177, 166
5022 data 176, 142,   6, 221, 169,  17, 141,  14
5023 data 221, 169,  81, 141,  15, 221,  96
```

Commodore 64: 6526 Time Of Day Clock

Mike Forani
Burlington, Ont.

Sometimes when you need it the most you can't get it. I'm referring to the time. The Commodore 64's TI$ function can sometimes cause a little frustration, too. You will often find that the value for TI$ will be off by a minute or two after having run for only 2 hours, and if you're using disk access in a program, you'll find the time will be off by even more. Therefore it would make a lot of sense to use the 6526's Time Of Day clock (TOD) inside the C64, so you can keep track of time more accurately. Ok, here we go!

The TI string is calculated on an interrupt basis: every time the interrupt routines are called its value is incremented by one. TI is measured in jiffies. A jiffy is 1/60 of a second and this is how often the interrupts occur. Sometimes people want to disable the stop key or the run/stop restore keys and in doing this you must reroute the interrupts. Depending on how you do this you could corrupt the TI$ timing. Using the TOD clock you can avoid this problem. The 6526 TOD clock does not depend on interrupts to update the time. The TOD clock is a free-running clock in the 6526 chip. Also if you are using machine language programming and you need to wedge into the interrupt routines you will often upset the TI$ timing and the time it gives you will be way off the mark. What is needed is something that is accurate and invisible to the C64. The TOD clock is about as invisible as you can get and as for accuracy it's as good as a $100 quartz watch. All you need to do is set the clock running and then you don't have to worry about it again. You can check the time as often or as little as you like.

Listed after this article is a machine language program that sets and reads the TOD clock. To use this along with basic you could have a little program like the following:

```
10 ifa=0thena=1:load "clock",8,1
20 sys 12*4096+15*256+3
30 input "enter time (hh,mm,ss)";b(2), b(1), b(0)
40 for a=0 to 2 : poke251+a, b(a) : next
50 sys 12*4096+15*256
```

What this program does is the following:

10 Load in the machine language routine that sets and reads the clock. It is located at $CF00-$CFFF. (52992-53247 in decimal)
20 Stop the clock if it is already running. (unwedge the program if it is wedged in.)
30 Get the time in a 24 hour format. (eg. 22,14,45)
40 Poke the time values into zero page for the machine language routine to use.
50 Call the machine language program to setup the clock and wedge into the interrupts so the code that displays the time will be executed.

This short little program just sets everything up. Once it is finished the clock will be displayed on the top right hand corner of the screen until you hit the run/stop restore or until you turn the clock off by using the 'sys' in line 20. All that will be going on at this point is that the time will be constantly displayed on the screen while the computer remains on. To get the time into a string incase you needed to use it for something you could try the following little routine:

```
10 a$="" : for a=32 to 39 : b(a-32)=peek(1024+a) : next
20 for a=0 to 7 : a$=a$+chr$(b(a)) : next
30 print a$
```

This routine does the following:

10 Get the time off the screen and into an array. (it is always displayed)
20 Turn the numbers in the array into characters and concatenate them into a string.
30 Print the string. (you could use it for something else)

In doing this the time can be taken off the screen and you can build your own TI$, or something like it, when you need it.

The machine language program has two entry points: $CF00 and $CF03. The first entry point sets up and starts the clock and the second stops it. Here is a description of what the program does:

Setup

1) Reconstruct the input, at locations $fb, $fc, $fd, so it is in the proper form for the 6526 TOD clock.
2) Set the clock values and start it running.
3) Wedge in at the IRQ vector and set a toggle flag. IRQ's occur 60 times a second and I do not need to update the clock display that often. 'Toggle' is used so the display is updated once every N interrupts. N can be changed, initially it is set to 15. Therefore the display is updated every 15/60ths of a second, or 4 times per second.
4) Return to BASIC and leave the clock to running.

If the clock is to be turned off the code is simply unwedged by replacing the wedged-in IRQ vector value with the normal IRQ vector value.

The interrupt-driven part of the machine language code works as follows:

Update

1) See if our toggle is timed down yet. If it isn't, go to the regular IRQ routines.
2) If the toggle is timed out then put the current character colour in the colour table in case it is a kernal 2/64 and the screen was cleared.
3) Get the hours register and check it for the am/pm bit, then get the minutes and the seconds.
4) Break apart the registers, so they can be displayed, and then display them. (ie. the register will read as *$23 and it must be broken down to *$32 and *$33 so it can be displayed.)
5) Go do the regular IRQ routines.

All of this will be occuring in background interrupts while you are operating in BASIC or machine language. Also listed is a BASIC loader for the machine language program if you do not have an assembler program.

In case you want the clock to be displayed differently, here are a couple of modifications you can make:

1) 12 hour clock. To do this you need to put NOP's in the machine language program at $cf8e, $cf8f, $cf90. In the BASIC program do the following pokes after the program is loaded in:

   poke 53134, 234 : poke 53135, 234 : poke 53136, 234

With these changes the clock will only display in a 12 hour mode. ie. 12:59:59 would roll to 01:00:00 instead of 13:00:00.

2) Rate of display. If for some reason you need to update and display the clock more than 4 times a second all you need to do is change the toggle value. In the machine language program change the $0f at $cf51 and $cf6a to a number between 1 and 255. For the basic program do the following pokes after the program is loaded into memory:

poke 53073, xx : poke53098, xx

Where xx is some number between 1 and 255. For the display rate change I would suggest that you try to keep the toggle values between 5 and 15. The reason for this is because you don't want to update the clock every interrupt or you will slow down the speed of the C64 and you must do it at least twice a second so the flash of the ':' can occur.

The reason for having the program reside at $CF00 is just so that it doesn't get in the way of anything, (I hope).

Enjoy this little program. I hope it helps you in discovering a little more about the workings of the Commodore 64. In my next article we could discuss the way in which to use the 6526 TOD clock to generate interrupts - AN ALARM CLOCK.

Editor's Note

When I tried Mike's clock display on my 64, the clock seemed to tick like a dripping faucet - steady for a while then two quick ones, or two short ones, etc. The problem? An inaccurate quartz crystal. (That 1 in a thousand had to be mine) But everything else works just fine and until I get the urge to write an extremely time sensitive program, I probably won't bother replacing it. If yours is running a little 'rough' too, and you need the accuracy, your service center can replace it in about 15 minutes (ie. 24 hours), or you hackers can wip one in yourself for around 5 bucks.

```
100             ;program variables
110  ;
120  irq        = $ea31          ;normal irq routines
130  cinv       = $0314          ;irq vector
140  scrn       = $0400          ;the screen starts here
150  colour     = $0286          ;current character colour
                                 value
160  ;
170  clrtab     = $d800          ;colour table ram
180  cia1       = $dc00          ;cia number 1 irq's
190  ;
200  secs       = $fb            ;seconds
210  mins       = $fc            ;minutes
220  hrs        = $fd            ;hours
230  ;
```

```
240          *=$cf00          ;the program resides at
                              $cf00 and up
250  ;
260  ;set up for set time
270  ;
280          jmp  setup       ;start the clock
290          jmp  kill        ;stop the clock
300  ;
310  setup   sei              ;no interrupts allowed
320          ldy  #2
330          sed              ;convert inputs to bcd
340  setl1   lda  #0
350          clc
360          ldx  secs,y
370  setl2   dex
380          bmi  setl3
390          adc  #1          ;adding one in decimal
400          bcc  setl2
410  setl3   sta  secs,y      ;this way a #23 will be-
                              come a #$23
420          dey
430          bpl  setl1
440          lda  hrs
450          cmp  #$13
460          bcc  setl4       ;check and see if pm. flag is
                              to be set
470          sec
480          sbc  #$12
490          ora  #128        ;set pm. bit
500          sta  hrs
510  ;
520  setl4   cld
530          ldy  hrs         ;get registers
540          ldx  mins
550          lda  secs
560  ;
570          sta  cia1+9      ;set the 6526's time of day
                              clock
580          stx  cia1+10     ;.a=seconds, .x=minutes
590          sty  cia1+11     ;.y=hours
600          lda  #0
610          sta  cia1+8      ;tenths of seconds - clock
                              starts here
620          lda  cia1+15
630          and  #%01111111  ;clock not alarm
640          sta  cia1+15
650  ;
660          lda  #<update
670          sta  cinv        ;wedge irq so i can
680          lda  #>update    ;update the clock
690          sta  cinv+1
700  ;
710          lda  #15         ;i don't want to do it every
720          sta  toggle      ;interrupt so use a toggle
730  ;
740          cli              ;enable the irq again
750          rts              ;return to basic
760  ;
770  kill    sei
780          lda  #<irq       ;stop displaying the clock
790          sta  cinv
800          lda  #>irq       ;put irq vector back
810          sta  cinv+1      ;to normal value
820          cli
830          rts              ;go back to basic
840  ;
850  ;update the clock and display it
860  ;
870  update  dec  toggle      ;decrement the toggle byte
880          bne  noupa       ;is it time to display the
                              clock yet
890          lda  #15         ;only need to update 4
                              times a second
900          sta  toggle      ;reset the toggle byte
910          lda  colour
920          ldy  #7          ;if a kernal 2 c64 then fix
                              up the colour
930  loop    sta  clrtab+32,y ;table values
940          dey
950          bpl  loop
960          lda  cia1+11     ;hours
970          bmi  tohere      ;see if am or pm
980          cmp  #$12
990          bne  okhere
1000         lda  #0          ;turned from 235959 to
                              000000
1010         beq  okhere
1020 tohere  and  #%00011111
1030         cmp  #$12        ;if pm then fix the hours
                              value
1040         bcs  okhere
1050         sei
1060         sed
1070         clc
1080         adc  #$12        ;it must be pm so add 12
                              hours to value
1090 okhere  sta  time+2
1100         cld
1110         cli
1120         lda  cia1+10     ;minutes
1130         sta  time+1
1140         lda  cia1+9      ;seconds
1150         sta  time
1160 ;
1170         ldx  #2
1180         ldy  #30
```

```
1190 goer   lda  time,x      ;print the hours then min-
                             utes
1200        sta  temp+1      ;and then the seconds
1210        jsr  distim
1220        iny
1230        iny
1240        sta  scrn,y
1250        lda  temp+1
1260        iny
1270        sta  scrn,y
1280        dex
1290        bpl  goer
1300 ;
1310        lda  cia1+8      ;tod tenths of seconds
1320        cmp  #5          ;see if we sshould print a ':'
1330        bcs  abov5       ;or a ' ' inbetween the
1340        lda  #58         ;hours/minutes/and sec-
                             onds
1350        .byt $2c         ;to 'hide' next lda
1360 abov5  lda  #32
1370        sta  scrn+37
1380        sta  scrn+34
1390 ;
1400 noupa  jmp  irq         ;go do normal irq stuff
1410 ;
1420 distim txa
1430        pha
1440        lda  #0          ;make the value in temp a
                             screen printable
1450        sta  temp        ;form
1460        lda  temp+1
1470        ldx  #3          ;take the 12 in one byte
                             and put
1480 disui  asl  a           ;it into two bytes 31 and 32
1490        rol  temp
1500        dex
1510        bpl  disui
1520        lda  temp+1
1530        and  #%00001111
1540        ora  #$30
1550        sta  temp+1
1560        pla
1570        tax
1580        lda  temp
1590        ora  #$30
1600        rts
1610 ;
1620 temp   *=*+2            ;temporary storage
1630 toggle *=*+1            ;toggle byte
1640 time   *=*+3            ;times stored here
1650 ;
1660        .end
```

```
1000 rem c64 time of day clock display
1010 for j= 52992 to 53239 : read x
1020 poke j,x : ch=ch+x : next
1030 if ch<> 31658  then print" checksum error " : end
1040 rem
1050 sys 52995
1060 input " enter time (hh,mm,ss) " ;b(2), b(1), b(0)
1070 for a=0 to 2 : poke 251+a, b(a) : next
1080 sys 52992
1090 end
1100 data  76,   6, 207,  76,  87, 207
1110 data 120, 160,   2, 248, 169,   0
1120 data  24, 182, 251, 202,  48,   4
1130 data 105,   1, 144, 249, 153, 251
1140 data   0, 136,  16, 238, 165, 253
1150 data 201,  19, 144,   7,  56, 233
1160 data  18,   9, 128, 133, 253, 216
1170 data 164, 253, 166, 252, 165, 251
1180 data 141,   9, 220, 142,  10, 220
1190 data 140,  11, 220, 169,   0, 141
1200 data   8, 220, 173,  15, 220,  41
1210 data 127, 141,  15, 220, 169, 100
1220 data 141,  20,   3, 169, 207, 141
1230 data  21,   3, 169,  15, 141, 250
1240 data 207,  88,  96, 120, 169,  49
1250 data 141,  20,   3, 169, 234, 141
1260 data  21,   3,  88,  96, 206, 250
1270 data 207, 208, 103, 169,  15, 141
1280 data 250, 207, 173, 134,   2, 160
1290 data   7, 153,  32, 216, 136,  16
1300 data 250, 173,  11, 220,  48,   8
1310 data 201,  18, 208,  15, 169,   0
1320 data 240,  11,  41,  31, 201,  18
1330 data 176,   5, 120, 248,  24, 105
1340 data  18, 141, 253, 207, 216,  88
1350 data 173,  10, 220, 141, 252, 207
1360 data 173,   9, 220, 141, 251, 207
1370 data 162,   2, 160,  30, 189, 251
1380 data 207, 141, 249, 207,  32, 211
1390 data 207, 200, 200, 153,   0,   4
1400 data 173, 249, 207, 200, 153,   0
1410 data   4, 202,  16, 232, 173,   8
1420 data 220, 201,   5, 176,   3, 169
1430 data  58,  44, 169,  32, 141,  37
1440 data   4, 141,  34,   4,  76,  49
1450 data 234, 138,  72, 169,   0, 141
1460 data 248, 207, 173, 249, 207, 162
1470 data   3,  10,  46, 248, 207, 202
1480 data  16, 249, 173, 249, 207,  41
1490 data  15,   9,  48, 141, 249, 207
1500 data 104, 170, 173, 248, 207,   9
1510 data  48,  96
```

JOYCURSOR:
A Cheap Mouse
For Your Commodore 64

Chris Zamara
Downsview, Ont.

If you program like me, you probably find that your fingers spend 90 percent of their time on just three keys on the keyboard – the shift, and the two cursor control keys. Let's face it: when you're staring at an incorrigible program trying to find out what's wrong with it, buzzing the cursor around the section of code you're contemplating seems to help. Also, if you use POWER, BASIC AID, or another utility that lets you scroll your program up and down with the cursor control keys, it's easy to while away the better part of a cup of coffee by scrolling the program up down up down up down until you can't see straight.

While doing this, I found myself wishing that I could move around the cursor without having to wear out my hand on those three poorly positioned keys. I wanted something external, something like the "mouse" used on some $10,000+ systems. Well, what's the equivalent of a mouse on a Commodore 64? Right! The ubiquitous joystick.

Run the loader program shown below. If you get a checksum error, re-check the DATA statement values and try again. When you get a successful RUN, JOYCURSOR will be enabled. With the joystick plugged into port #2, you should be able to move the cursor around in all directions, including diagonally. You can change the speed that the cursor moves by POKEing a different value in location 49177 (it is originally set to 5).

Use RUN/STOP RESTORE to disable JOYCURSOR, and SYS 49152 to re-enable it. enJOY!

```
100 rem * data loader for "JOYCURSOR" *
110 :
120 cs=0    : rem * checksum *
130 os=49152 : rem * object start
140 :
150 read b : if b<0 then 180
160 cs = cs + b
170 poke os,b: os=os+1: goto150
180 :
190 if cs <> 12839 then print" * checksum error
* " : end
200 :
210 sys 49152 :rem * enable "JOYCURSOR"
220 print" ** Ok, JOYCURSOR is enabled. ** "
230 :
240 end
250 data 120, 169,  18, 141,  20,   3
260 data 169, 192, 141,  21,   3,  88
270 data  96, 145,  17,  29, 157,   0
280 data 238,  17, 192, 173,  17, 192
290 data 201,   5, 208,  85, 169,   0
300 data 141,  17, 192, 173,   0, 220
310 data 201, 127, 240,  73, 169,   1
320 data  44,   0, 220, 208,   6, 173
330 data  13, 192,  32,  95, 192, 169
340 data   2,  44,   0, 220, 208,   6
350 data 173,  14, 192,  32,  95, 192
360 data 169,   4,  44,   0, 220, 208
370 data   6, 173,  16, 192,  32,  95
380 data 192, 169,   8,  44,   0, 220
390 data 208,   6, 173,  15, 192,  32
400 data  95, 192,  76, 113, 192, 166
410 data 198, 157, 119,   2, 230, 198
420 data 165, 198, 201,  10,  48,   4
430 data 169,   0, 133, 198,  96,  76
440 data  49, 234,  -1
```

An Executive SX–64 Emulator

Jim Butterfield
Toronto, Ont.

Can't get an SX–64, because you can't afford one or they are out of stock? And you say you need to check out a program or two to see if it works OK on the SX–64 as well as on your regular 64?

This procedure will convert your Commodore 64 into a logical SX–64. It replaces the ROM and a little of the RAM with SX–64 information. Thus, you can try your hand at running the machine.

Not Much Difference

In fact, the SX–64 is very close to a Commodore 64. The major differences are: absence of cassette tape; different background/foreground colors; a redefinition of the RUN/STOP key; and reinstatement of easy screen POKEs. The screen POKE feature, together with a few minor cleanups, is in all new Commodore 64 units; but if your machine dates back a few months or more, it will be new for you.

Commodore have very carefully preserved "entry points" in the computer's logic. Almost anything you code – machine language or BASIC – will still work on the 64. We can all think of system features that we would have liked to see changed or added, but Commodore have stayed away from most of them. As a result, there's excellent compatibility between portable and regular 64.

BASIC is identical to that of previous units; in fact, it hasn't changed since VIC–20 days. Even though BASIC is the same, the procedure given below loads it in; that way, future changes may be accomodated.

How To Write It

Obtain access to an SX–64. Bring along a disk and format it. Now: enter the following program:

```
100 data 1024,2023
110 data 55296,56295
120 data 40960,49151
130 data 57344,65535
140 for j = 1 to 8
150 read x:t = t + x
160 next j
170 if t<>327628 then stop
180 restore
190 for j = 1 to 4
200 read x,y
210 open 1,8,3, "0:sx" + str$(j) + ",p,w"
220 x% = x/256:z = x-x%*256
230 print#1,chr$(z);chr$(x%);
240 for k = x to y
250 print#1,chr$(peek(k));
260 next k
270 close 1
280 next j
```

Be sure to include the semicolons at the end of lines 230 and 250. When it's ready, RUN the program. It will take some time, but eventually four program files will be written on your disk. Return the SX–64 to its owner and take your disk home.

How To Read It

Power up your Commodore 64. Important: if possible, disconnect external devices such as C–Link or Buscard. Enter the following program:

```
90 poke 53280,3:poke 53281,1
100 a = a + 1:if a = 5 goto 120
110 load "sx" + str$(a),8,1
120 print chr$(31);
130 poke 1,53
```

Run the program. When it's finished, you'll have a pseudo–SX–64. Check it by commanding, LOAD "ANYTHING" – the computer will reply DEVICE NOT AVAILABLE. The SX–64 doesn't have tape.

How It's Done

We write four blocks: screen, color nybbles, BASIC ROM, and kernal ROM. We write them as program files, so the first two bytes are the load address. By the way, you couldn't save the Kernal ROM from a typical machine language monitor, since there's no way you could fit in that last address of 65535 (or $FFFF). In this case, Basic seems to have a slight advantage over the monitor. We must set the screen background and border colors separately, as well as the cursor color, since they are not stored within any of the four areas mentioned.

The reading program is elegant, but hard to read if you don't know the trick. Here it is: after BASIC performs a LOAD, it always returns to the first statement. Thus, there's really an invisible loop from line 110 back to 90. When all loads are finished, variable A equals 5 and we skip ahead to set the cursor color.

Now: we've been reading this ROM information into RAM memory. But on the 64, we can switch ROM out and let RAM take over. If we wanted to do this for just BASIC, we'd give POKE 1,54; for both BASIC and Kernal, we must say POKE 1,53.

Using the same methods, you can switch logic between various generations of the Commodore 64.

Conclusion

If you don't have access to an SX–64, there will be a disk in the TPUG library to do the job for you.

Now you have an SX–64, at least in a logical sense. As I said before, you won't find much difference from the Commodore 64. But at least you'll know how it feels. Now, if it only had a handle. . .

The Transactor
The Tech/News Journal For Commodore Computers

PAYS

$40

per page for articles

We're also looking for professionally drawn cartoons!

Send all material to:

The Editor
The Transactor
500 Steeles Avenue
Milton, Ontario
L9T 3P7

Volume 5 Editorial Schedule

Issue#	Theme	Copy Due	Printed	Release Date
1	Graphics and Sound	Feb 1	Mar 19	April 1
2	The Transition to Machine Code	Apr 1	May 21	June 1
3	Software Protection & Piracy	Jun 1	Jul 23	August 1
4	Business and Education	Aug 1	Sep 17	October 1
5	Hardware and Peripherals	Oct 1	Nov 19	December 1
6	Programming Aids & Utilities	Dec 1	Jan 19	February 1/85

Volume 6 Editorial Schedule

1	Communications & Networking	Feb 1	Mar 21	April 1/85
2	Languages	Apr 1	May 20	June 1
3	Implementing The Sciences	Jun 1	Jul 18	August 1
4	Hardware & Software Interfacing	Aug 1	Sep 21	October 1
5	Real Life Applications	Oct 1	Nov 19	December 1

Advertisers and Authors should have material submitted no later than the 'Copy Due' date to be included with the respective issue.

PRO·LINE SOFTWARE

A CANADIAN COMPANY

designing, developing, manufacturing, publishing and distributing microcomputer software

DEALER ENQUIRIES WELCOME
AUTHOR'S SUBMISSIONS INVITED
CALL OR WRITE

(416) 273-6350
PRO·LINE SOFTWARE
755 THE QUEENSWAY EAST, UNIT 8,
MISSISSAUGA, ONTARIO L4Y 4C5

GRAPH-TERM 64
A GRAPHICS TERMINAL PROGRAM FOR THE COMMODORE-64

GRAPH-TERM 64 is a 100% machine-language program which

- plots hi-res graphs generated by a mainframe computer or the C-64 in standard Tektronix® format
- downloads text (36K) or plot files (20K)
- creates instant replays of text or graphs at high speed, slow motion or stop action
- creates hard copies of plots on the Commodore 1520 Plotter

In addition, the machine language subroutines used in **GRAPH-TERM 64** are documented so you can use them in your own programs to create fast, compact plot files and to drive the plotter at top speed.

VISA **$49.95** U.S. master charge
TO ORDER

Specify disk or tape
Add $4.00 postage and handling for U.S. and Canada
Other foreign orders add 20%
Michigan residents add 4% sales tax

BENNETT SOFTWARE CO.
3465 Yellowstone Dr.
Ann Arbor, MI 48105
(313) 665-4156
Dealer inquiries invited
The 1520 plotter and the Commodore 64 are products of Commodore Business Machines.

The Intelligent Software Package

For $35, you get all this on one disk:

DATA BASE: A complete fixed record-length data base. Sort on any key, select using full logical operators on any key or keys, perform numeric manipulation on fields. All fields in a record fully customizable. Screen editing for records. Can be used for accounts-receivable, inventory control, or as an electronic rolodex. If you use your Commodore for nothing else, this program will justify its expense.

WORD PROCESSOR: A full-featured word processor: very fast file commands (including disk file catalog), screen editing, string searches, full control over margins, spacing, paging, and justification (all commands imbedded in text). A very powerful, easy-to-learn program. Includes a program interfacing W/P with DATA BASE to create custom form letters.

SPREADSHEET: Turns your Commodore into a visible balance sheet. Screen editing. Great for financial forecasting.

BASEBALL MANAGER: Compiles batting statistics for a baseball or softball league. Generates reports on a player, team, or the entire league (including standings).

All programs will load and run on any and every Commodore computer having a minimum of 10k RAM; all programs fully support tape, disk, and printer. Any two programs on cassette, $20. Price includes shipping within USA and Canada; Calif. residents add 6%. For orders over 10 in quantity, deduct 35%.

Since this ad is the catalog, no response to inquiries will be made; however, documentation for any one program may be purchased separately for $2 postpaid (deductible from later order). Thank you.

William Robbins, Box 3745, San Rafael, CA 94912

Why Blank "Cheat" Sheets?

Because They're Better Blank

O.K. So now you've got the best Commodore 64 in the world, and lots of complex software to run on it. One problem. Unless you work with some of these programs everyday or are a computer genius, who can keep all those commands straight? "F5" in one program means one thing, and "F5" in another program means something else. A few companies do offer a solution — a die cut "cheat" sheet that attaches to your keyboard with all the commands of one program printed on it. Great idea, unless you need them for 10 or 20 programs. You could purchase another disk drive for the same investment. Our solution? Simple. A pack of 12 lined cards, die cut to fit your keyboard and just waiting to be filled with those problem commands **you** forget most often. Simple? Yes, but effective. Now you can have **all** your program commands right at your finger tips on YOUR VERY OWN, custom designed "cheat" sheets. Order a couple packs today!

Please send me the following: Prices in U.S. dollars

Qty.	Item	Price
_____	Sets of 12 C-64 Keyboard Cheat Sheets @ $15.95	$ _____
_____	2 Packs (24 Sheets) for $24.95	$
Total for Merchandise Shipping and Handling		$ 2.00
Canadian Funds Surcharge		$ 3.00
	TOTAL ENCLOSED	$ _____

☐ Please Charge to: ☐ MasterCard ☐ VISA

Number _____ Expires _____

SHIP TO: Name _____

Address _____

City _____

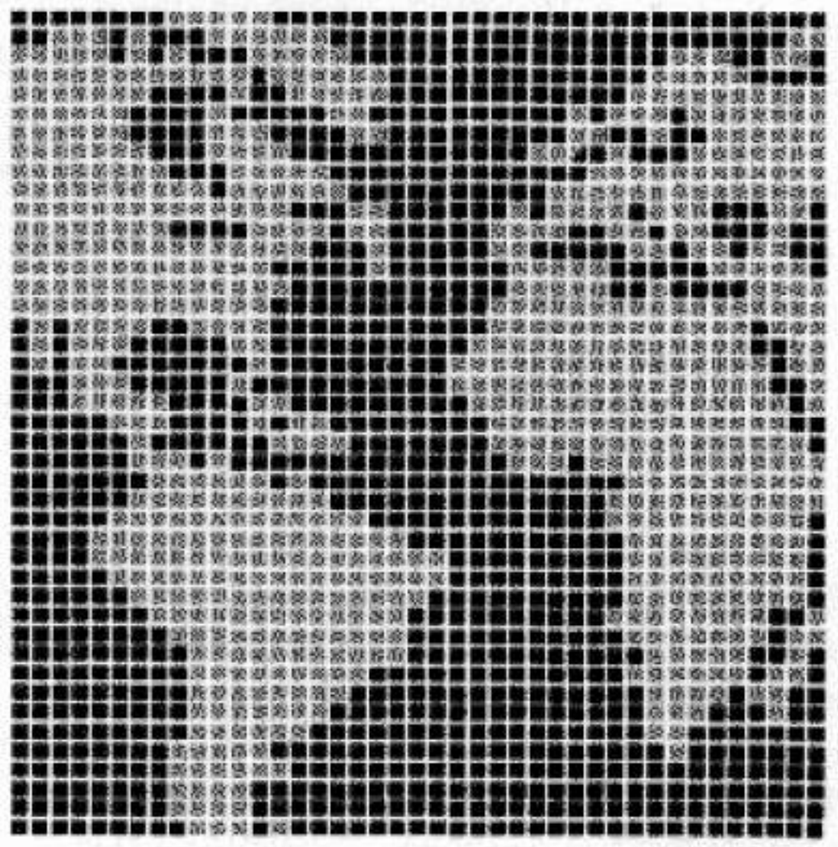
State/Zip _____

Dealer Inquiries Invited

Bytes & Pieces, Inc.

550 N. 68th Street
Wauwatosa, WI 53213
414/257-3562

INTERNATIONAL CENTRE, TORONTO
NOVEMBER 29 & 30, DECEMBER 1 & 2, 1984



THE WORLD OF COMMODORE II

The Company that had the foresight and imagination to design and build more computers for home, business and education than any other will be presenting the most farsighted and imaginative show to date with exhibitors from around the World.

The 1983 Canadian World of Commodore Show was the largest and best attended show in Commodore International's history. Larger than any other Commodore show in the World and this year's show will be even larger.

World of Commodore II is designed specifically to appeal to the interests and needs of present and potential Commodore owners.

Come and explore the World of Commodore.

world of commodore II

A HUNTER NICHOLS PRESENTATION.
FOR MORE INFORMATION CALL
DEBBIE BANNON
(416) 439-4140

(Magreeable™

software

STOCK HELPER™

Commodore 64 and VIC-20

Stock HELPER is a tool to maintain a history of stock prices and market indicators on diskette, to display charts, and to calculate moving averages. Stock HELPER was designed and written by a "weekend investor" for other weekend investors.

Stock HELPER is available on diskette for:

Commodore 64	$30.00	($37.00 Canadian)
VIC-20 (16K)	$27.00	($33.25 Canadian)

plus $1.25 shipping ($1.55 Canadian)

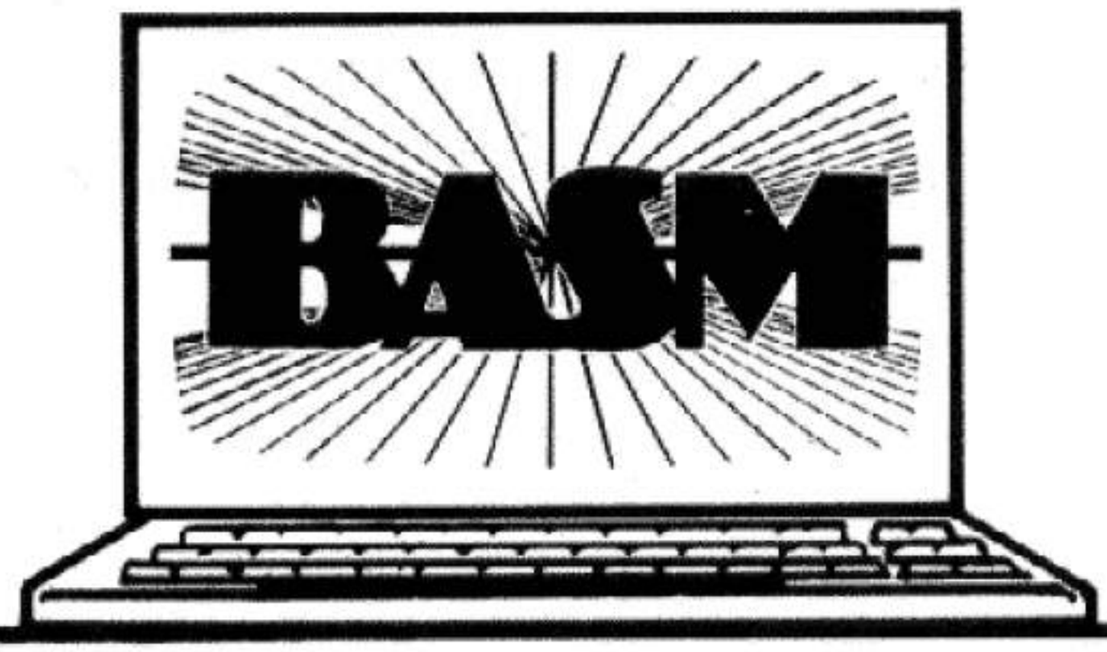
The VIC-20 version only charts 26 bi-weekly periods rather than 52 weekly periods.

(M)agreeable software, inc.

5925 Magnolia Lane • Plymouth, MN 55442

(612) 559-1108

(M)agreeable and HELPER are trademarks of (M)agreeable software, inc. Commodore 64 and VIC-20 are trademarks of Commodore Electronics Ltd.



The computer language for the New Professional.

BASM is a unique blend of BASIC and standard 6510 Assembly language. This ingenious combination of familiarity and flexibility provides an easier transition to Assembly language while cutting your programming time by 75%! Your program will then run over 200 times faster than Commodore BASIC!

BASM—an entirely new programming environment for the Commodore 64. (Also available for Atari)

TO ORDER WRITE OR PHONE

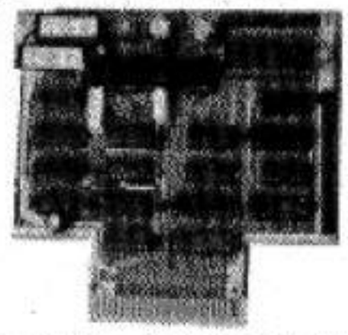
COMPUTER ALLIANCE

10730 White Oak Avenue
Granada Hills, CA. 91344
(818) 368-4089

GET CONTROL WITH YOUR C64

A/D CONVERTER

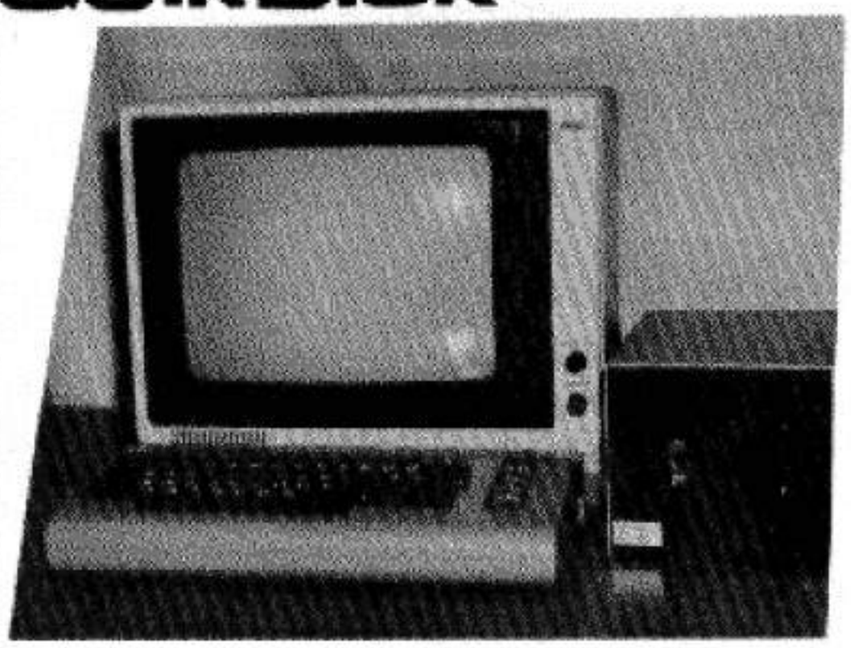
DIGITAL I/O



* ANALOG TO DIGITAL CONVERTER - 16 Channel 12 Bit, 0-10 Volt Inputs,Dual slope with up to 30 conversion per second. Controlled from Basic or Machine language
* 12 BIT DIGITAL TO ANALOG CONVERTER- 0-10 volt output, drive chart recorder, or control the speed of a D.C. motor drive.
* 12 DIGITAL INPUTS - TTL Compatible. Monitor Switches, Contacts, Fire alarms, or Burglar alarms
* 10 DIGITAL OUTPUTS- TTL Compatible. Control relays, Motors, turn on alarms. Control sprinklers
* C-MOS REAL TIME CLOCK CALANDER- with battery backup - set it once and forget it. Basic and machine language programs provided for operation entry and screen output. DY/MO/YR HR/MN/SC

THE DATAONE comes complete with user's manual and software driver
Assembled and tested $249.00
Add $5.00 for shipping USA

QUIKDISK



IF YOU NEED RELIABLITY AND SPEED then QUIKDISK is the answer. QUIKDISK is a high performance floppy disk system designed especially for the Commodore 64 series computer. It is part of the PBDISK series of floppy disk systems and is optimized to provide extremely high speed and reliable operation. The QUIKDISK system consists of a small disk controller module, a cable assembly, and a standard disk drive assembly. The controller will interface to three inch, five and one quarter inch, or eight inch disk drives. The QUIKDISK controller module plugs into the cartridge slot of the computer and a flat ribbon cable connects to the drive.

PDOS software emulates a Commodore disk drive by intercepting the disk commands from the machine. QUIKDISK operates, however, bbby transferring data directly from the diskette to the computer memory. With a data transfer rate of 250,000 bits per second, over ten times faster than the serial bus, QUIKDISK provides emulation at the fastest possible speed. A full set of disk utilities are also available.

Controller card with software	$ 295.00
Model C877-1 single 8" system	$1095.00
Model C540-1 single 5" system	$ 595.00
Model C340-1 dual 3.5" system	$ 995.00

Dealer inquiries accepted

KMMM PASCAL

KMMM PASCAL for Commodore C64 by Wilsery one of the newest high level languages, KMMM PASCAL is a true compiler that generates machine code from pascal source... FAST! Editor,Compiler and Translatorincluded. KMMM is a subset of Jensen and Wirth Pascal.

PRICE.....$99.00

FOR INFORMATION, SEE YOUR DEALER OR:

MICROTECH

P.O. BOX 182 • LANGHORNE, PA 19047 • (215) 757-0282

"PET IS A TRADEMARK OF COMMODORE AND IS A TRADEMARK OF ROCKWELL.

The Reference Transactor

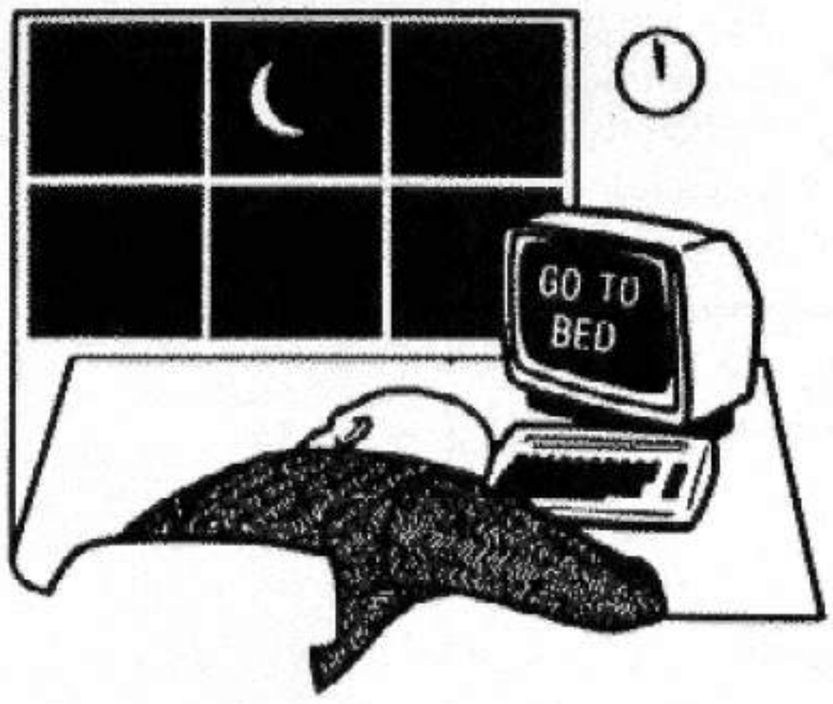
Coming This Fall!

See News BRK for details

The MIDNITE SOFTWARE GAZETTE

The PAPER

Five years of service to the PET community.



The Independent U.S. magazine for users of Commodore brand computers.

EDITORS: Jim and Ellen Strasma
Sample issue free on request, from:

635 MAPLE □ MT. ZION, IL 62549 USA

MICRO-FAX

54 FLERIMAC ROAD, WEST HILL, ONTARIO M1E 4A9 CANADA
TELEPHONE: (416) 282-1532

THE 64 SOFTWARE HOUSE

ENTERTAINMENT

A.E.	"BRODERBUND"	D	39.95
SPARE CHANGE	— " —	D	39.95
OPERATION WHIRLWIND	— " —	D	40.95
ZAXXON	"SYNAPSE"	T/D	44.95
QUASIMOTO	— " —	T/D	39.95
SHAMUS CASE II	— " —	T/D	39.95
SORCERER	(INFOCOM)	D	59.95
FLIGHT SIMULATOR	(SUBLOG)	T/D	59.95
SAMMY LIGHTFOOT	(SIERRA)	D	33.95
PAINT BRUSH	(HES)	C	23.95
BRUCE LEE	(DATA SOFT)	C/D	39.95
CASTLE WOLFENSTEIN	(MUSE)	D	33.95

BUSINESS

HOME ACCOUNTANT	(CONT)	D	76.95
VIP TERMINAL		D	65.95
ELECTRONIC CHECKBOOK	(TIMEWORKS)	T/D	27.95
BANK STREETWRITER		D	72.95

SPECIALS

KOALA PAD AND PAINTER & SLICK STICK (JOYSTICK)		109.95
WHIZ KIDS INTRO TO BASIC	D	29.95
STRIP POKER	D	37.95
STAR MAZE	D	38.95
INTRODUCING THE RITEMAN PRINTER 120 CPS 1 YR. WARRANTY		499.00

ONTARIO RESIDENTS ADD 7% SALES TAX.
ORDERING & TERMS: SEND CASHIER CHECK, MONEY ORDER, OR CERTIFIED CHECK. VISA/MASTERCARD PLEASE INCLUDE CARD NUMBER & EXPIRY DATE AND SIGNATURE.
ADD $2.50 FOR SHIPPING AND HANDLING.
ALL ITEMS SUBJECT TO AVAILABILITY. PRICES SUBJECT TO CHANGE WITHOUT NOTICE.
FOR CATALOG SEND $1.50 REDEEMABLE.

INFOMAG INC.

	DISCOUNTED PRICE FOR MOST SYST.
APPLE, ATARI, C-64, VIC 20	
BRODERBUND *(GAMES)*	
Lode Runner D C	$41
Spare Change D	$41
Drol D	$41
Choplifter C D	$41
Seafox D C	$36
(Cartridge version extra)	
Bank Street Writer D	$85
INFOCOM *(ADVENTURES)*	
Witness D	$59
Planetfall D	$59
SYNAPSE *(ATARI & C-64, GAMES)*	
Fort Apocalypse D T	$41
Blue Max D T	$41
SIRIUS *(GAMES — for most)*	
Snake Byte D	$36
Bandits D	$41
Type Attac D	$47
Squish'em C APPLE	$48
D C-64	$41
SMA (SYSTEMS MGT. ASSOC.)	
Documate-template C-64	$16
Code pro-64 — Tutorial for basic plus sprite & music gen.	$70
COMM* DATA EDUCATIONAL (VIC & C-64)	
Toddler Tutor	$34
Primary Math Tutor	$34
Math Tutor	$34
English Invaders Games	$34
Gotcha Math Games	$34
Dealer inquiries for:	
Programmer's Institute	
Kiwisoft	
Victory Software	
Comm* Data	
SMA	

C-64, VIC, ATARI, TRS-COLOR PROGRAMMER'S INSTITUTE FUTUREHOUSE)	
"Edumate Light Pen"	$36
C-64, VIC, Atari	
"Playground Software" t.m. (Uses Light Pen) C-64 & Atari	
Animal Crackers D	$36
Computer Crayons D	$36
Alphabet Arcade D	$36
Bedtime Stories D	$36
"C.P.A. Complete Personal Account" t.m. C-64, VIC, TRS, Color, Atari	
Complete Set (1, 2 & 3) DT	$94
Finance #1 D T	$36
Finance #2 D T	$36
Finance #3 D T	$36
Finance #4 D T	$36
KIWISOFT (C-64)	
Paintpic-64	$45
Art on your screen	
VICTORY SOFTWARE	
20/64 Dual Packs	
Cassettes (T) or Disks (D)	
GAMES	
Metamorphosis T D	$30
Creators Revenge T D	$30
Labyrinth of Creator T D	$30
Galactic Conquest T D	$30
Kongo Kong T D	$30
Chomper Man T D	$30
Annihilator T D	$30
Adventure Pack I (3 Prog) T D	$30
Adventure Pack II (3 Prog) T D	$30
Bounty Hunter (Adv) T D	$30
Grave Robbers-Graphic (Adv) T D	$24
(Disk version: $4. extra)	
PRECISION SOFTWARE (SILICOM INT'L)	
Super Base 64 Data Management System D	$117
Calc Result (Easy)	$108
Calc Result (Advanced)	$202
(C) Cartridge (T) Tape (D) Diskette	

Please call for info on your computer model, availability and specific price. Send certified cheque, money order or call and use your visa or Mastercard. Personal cheques require two or three weeks to clear. All prices subject to change without notice. Please include $2.00 per order for postage and handling. Quebec residents only add P.s.T.

Call Toll Free 1-(800)361-0847
except Western Canada, Nfld. and Montreal area (514) call collect
CALL COLLECT (514) 325-6203

between 9 a.m. and 5 p.m. Eastern time
or send order to: 6864 JARRY EAST, MONTREAL, QUE. H1P 3C1

This Space Could Be Transacting For You!

Kelly M. George
Advertising Manager
416 876 4741

COMMODORE OWNERS WE'LL CHECK YOU OUT

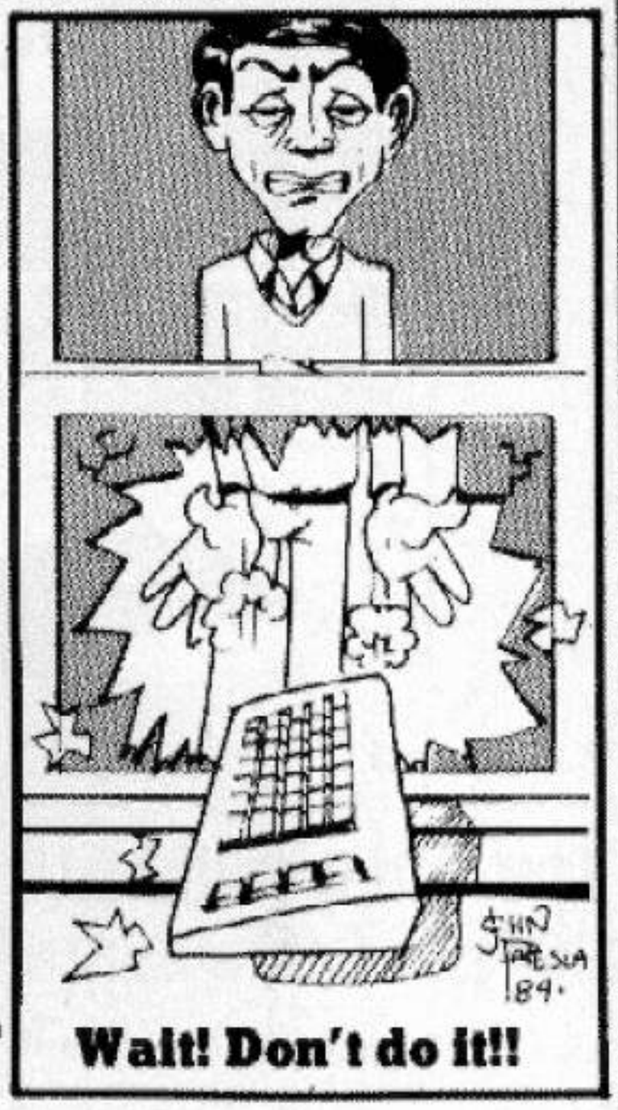
Mr Tester TM
Is your Commodore 64 TM Disk Drive, Printer, Memory, Joystick, Monitor and Sound Chip operating correctly?

You may never know for sure. Mr. Tester is a complete diagnostic that tests:

1.) Full joystick operation in all axis .
2.) Continuous or standard comprehensive memory test.
3.) CommodoreTM SID chip test for sound analysis.
4.) Screen alignment and color test.
5.) Complete read/write Disk Track and Block Test.
6.) Diskette format analysis to check Floppys.
7.) Complete printer test.
8.) Complete keyboard test.
9.) Cassette read/write test.

All this for only

$29.95



order from

M-W Dist. Inc.
1342B Route 23
Butler, N.J. 07405
201-838-9027

COMMODORE OWNERS WE'LL FIX YOUR FILES WITH FANTASTIC FILER™

The all purpose Data Base management system that provides:

1.) Menu driven subsections
2.) Logical key functions
3.) Average of 1000 records per disk
4.) Fast record access time
5.) Search for records by record number or by specific search criteria
6.) Easy to edit, delete or update records
7.) Interface with **FANTISTIC FORMS™** to print mailing labels or columnar reports
8.) Complete reference manual
9.) Technical support available to answer questions
10.) Up to 255 characters per record and up to 15 fields

All this for only

$29.95

available from:

Micro-W. D.I.
P.O. Box 113
Butler, N.J. 07405
(201) 838-9027

Commodore 64
and
VIC-20

SuperTerm

$149.95

Telecommunications with a difference!

Unexcelled communications power and compatibility, especially for professionals and serious computer users. Look us over; **SuperTerm** isn't just "another" terminal program. Like our famous Terminal-40, **it's the one others will be judged by.**

- **EMULATION**—Most popular terminal protocols: cursor addressing, clear, home, etc.
- **EDITING**—Full-screen editing of Receive Buffer
- **UP/DOWNLOAD FORMATS**—CBM, Xon-Xoff, ACK-NAK, CompuServe, etc.
- **FLEXIBILITY**—Select baud, duplex, parity, stopbits, etc. Even work off-line, then upload to system!
- **DISPLAY MODES**—40 column; 80/132 with side-scrolling
- **FUNCTION KEYS**—8 standard, 52 user-defined
- **BUFFERS**—Receive, Transmit, Program, and Screen
- **PRINTING**—Continuous printing with Smart ASCII interface and parallel printer; buffered printing otherwise
- **DISK SUPPORT**—Directory, Copy, Rename, Scratch

Options are selected by menus and EXEC file. Software on disk with special cartridge module. **Compatible with CBM and HES Automodems**; select ORIG/ANS mode, manual or autodial.

Write for the full story on SuperTerm; or, if you already want that difference, order today!

Requires: Commodore 64 or VIC-20, disk drive or Datasette, and compatible modem. VIC version requires 16K memory expansion. Please specify VIC or 64 when ordering.

Smart ASCII Plus . . . $59.95

The only interface which supports streaming —sending characters simultaneously to the screen and printer — with SuperTerm.

Also great for use with your own programs or most application programs, i.e., word processors. **Print modes:** CBM Graphics (w/many dot-addr printers), TRANSLATE, DaisyTRANSLATE, CBM/True ASCII, and PIPELINE.

Complete with printer cable and manual. On disk or cassette.

VIC 20 and Commodore 64 are trademarks of Commodore Electronics, Ltd.

(816) 333-7200

Send for a free brochure.

MIDWEST MICRO inc.

MAIL ORDER: Add $1.50 shipping and handling ($3.50 for C.O.D.); VISA/Mastercard accepted (card# and exp. date). MO residents add 5.625% sales tax. Foreign orders payable U.S.$, U.S. Bank ONLY; add $5 shp/hndlg.

311 WEST 72nd ST. • KANSAS CITY • MO • 64114

COMMODORE
-USER WRITTEN SOFTWARE-

Supporting all COMMODORE computers
Written by users, for users
★ GAMES ★ UTILITIES ★ EDUCATIONAL ★

VIC 20™

Vic 20 collections #1, 2, 3, 4, 5, 6
over 70 programs per collection-Tape/Disk - $10.00
Vic 20 collections #7, 8
over 50 programs per collection - Tape/Disk -$10.00

COMMODORE 64™

64 collections #1, 2, 3, 4, 5, 6, 7,
over 25 programs per collection - Tape/Disk - $10.00

PET® / CBM®

22 collections - Tape/Disk - $10.00

DINSET™: Reset Switch

Works on Vic 20 or Commodore 64 – $5.00

SERIAL CABLES

10Ft.—$10.00 15Ft.—$15.00

LOC-LITE™

Operation Status Indicator Assembled & Tested
$20.00

All prices include shipping and handling.
CHECK, MONEY ORDERS,
VISA and MASTERCARD accepted.
For A Free Catalog Write:

Public Domain, Inc.

5025 S. Rangeline Rd., W. Milton, OH 45383
10:00 a.m. - 5:00 p.m. EST – Mon thru Fri.
(513) 698-5638 or (513) 339-1725

VIC 20™, CBM™ and Commodore 64™ are Trademarks of Commodore Electronics Ltd
PET® is a Registered Trademark of Commodore Business Machines, Inc

THE BANNER MACHINE™

Menu-driven program works like a word processor. Great for businesses, schools, or organizations. Produces large signs up to 13" tall by any length. Make borders of widths up to ¾". Eight sizes of letters from ¾" to 8" high. Proportional spacing, automatic centering, right and left justification. Use with Gemini 10 or 10X; Epson MX with Graftrax, or the RX or FX; Commodore 1525E or MPS 801; and the Banana. Four extra fonts available ($19.95 each). Tape or disk $49.95

Menu Driven Disk Operating System Execute disk commands by reading the menu and pressing just one key: LOAD, SAVE, initialize disk, validate, scratch, rename, COPY, auto list, renumber, search, replace, and more! Disk $29.95

Flex File 2.1 By Michael Riley. Save up to 1500 typical records on a 1541 disk drive. Print information on labels or in report format. Select records 9 ways. Sort on up to 3 keys. Calculate report columns. 1541 · 4040 · 2031 Disk $59.95

CTRL-64 Permits listing of C-64 programs on non-Commodore printers. Lists control symbols in readable form. Disk $24.95

Screendump Print a copy of the C-64 screen simply by pressing just two keys. This machine-language program is compatible with most software. $19.95

Chessmate 64 Analyze your own games, master games, book games, and openings. Save, print, and watch your games in a unique "chess movie." Memorize any board position and recall it after you have played through variations. Disk $29.95

Formulator A formula scientific calculator for tasks which require repetitive arithmetic computations. Save formulas and numeric expressions. Ideal for chemistry, engineering, or physics students. Tape or disk $39.95

Space Raider An amazing arcade simulation. Your mission is to destroy the enemy ships. $19.95

Order Toll Free: 800-762-5645
Information: 703-491-6502
HOURS: 10 a.m. to 4 p.m. Mon-Sat

Cardinal Software™

13646 Jeff Davis Hwy.
Woodbridge, VA 22191

Catalogs available.
Specify: Educational, Business/Utilities, or Games/Simulations.

Commodore 64 is a registered trademark of Commodore Electronics Ltd.

Let The SMART 64 Terminal

COMMODORE 64* Do The DRIVING

No matter which direction you wish to travel in, experience the advantage of computer communications with The SMART 64 Terminal. Discover the program that puts you on the Right Road to: Public-Access Networks, University Systems, Private Company Computers and Financial Services.

The SMART 64 Terminal designed with Quality-Bred features, Affordable Pricing . . . And Service.

So why not travel the communications highways the SMART way!

Suggested **$39.95** Retail

Accessories included:

- ☐ *Selective Storage of Received Data.*
- ☐ *Alarm Timer.*
- ☐ *40 or 80 Col. Operation*.*
- ☐ *Auto-Dial.*
- ☐ *User-Defined Function Keys, Screen Colors, Printer and Modem Setting.*
- ☐ *Screen Print.*
- ☐ *Disk Wedge Built-In!*
- ☐ *Formatted Lines.*
- ☐ *Review, Rearrange, Print Files.*
- ☐ *Sends/Receives Programs and Files of ANY SIZE.*
- ☐ *Adjustable transmit/receive tables allow custom requirements. These and other features make The SMART 64 Terminal the best choice for grand touring telecommunications.*

*Commodore 64 registered trademark of Commodore Business Machines Inc

*Supports 80-column cartridge by Data 20 Corporation

Dealer Availability
Call (203) 389-8383

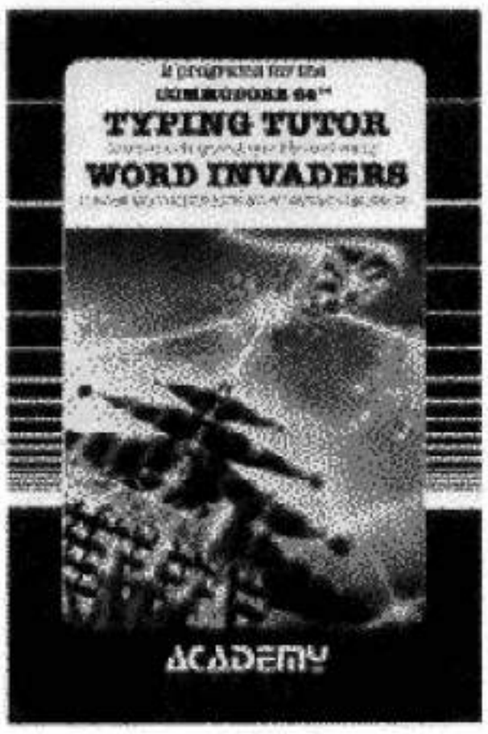
MICROTECHNIC® SOLUTIONS
P.O. BOX 2940, New Haven, Ct. 06515

Prices are in US dollars.

VIC 20™ COMMODORE 64™

JOIN THE COMPUTER REVOLUTION WITH A MASTERY OF THE KEYBOARD!

In the age of the computer, *everyone* from the school child to the Chairman of the Board should be at home at the computer keyboard. Soon there will be a computer terminal on every desk and in every home. Learn how to use it right ...and have some fun at the same time!



Rated THE BEST educational program for the VIC 20™ by Creative Computing Magazine

TYPING TUTOR PLUS WORD INVADERS

The proven way to learn touch typing.

COMMODORE 64 Tape $21.95 COMMODORE 64 Disk $24.95
VIC 20 (unexpanded) Tape $21.95

Typing Tutor plus Word Invaders makes learning the keyboard easy and fun! Typing Tutor teaches the keyboard in easy steps. Word Invaders makes typing practice an entertaining game. Highly praised by customers:

"Typing Tutor is great!", "Fantastic", "Excellent", High quality", "Our children (ages 7-15) literally wait in line to use it.", "Even my little sister likes it", "Word Invaders is sensational!"

Customer comment says it all . . .

". . . it was everything you advertised it would be. In three weeks, my 13 year old son, who had never typed before, was typing 35 w.p.m. I had improved my typing speed 15 w.p.m. and my husband was able to keep up with his college typing class by practicing at home."

NEW!

IFR (FLIGHT SIMULATOR) CARTRIDGE FOR THE VIC 20

$39.95

COMMODORE 64 TAPE OR DISC $29.95

JOYSTICK REQUIRED

Put yourself in the pilot's seat! A very challenging realistic simulation of instrument flying in a light plane. Take off, navigate over difficult terrain, and land at one of the 4 airports. Artificial horizon, ILS, and other working instruments on screen. Full aircraft features. Realistic aircraft performance — stalls/spins, etc. Transport yourself to a real-time adventure in the sky. Flight tested by professional pilots and judged "terrific"!

VISA Shipping and handling $1.00 per order. CA residents add 6% tax. MasterCard

ACADEMY SOFTWARE
P.O. Box 9403, San Rafael, CA 94912 (415) 499-0850

Programmers: Write to our New Program Manager concerning any exceptional VIC 20™ or Commodore 64™ game or other program you have developed.

MICROCOMPUTER SUPPLIES

Call us for all your C-64 Software

Memorex SS/DD	**$33.00/10**
ECtype SS/DD	**25.00/10**
Single Superdrive	**650.00**
Dual Superdrive	**1050.00**
Games on Disk	
Spy's Demise	**39.95**
Pensate	**39.95**
Thunder Bombs	**39.95**
J-Bird (Best Arcade)	**49.95**
Flight Simulator	**37.95**
Business on Disk	
Personal Accountant (Best)	**49.95**
Multiplan	**124.95**
Data Manager	**31.95**
Electronic Checkbook	**31.95**
Money Manager	**31.95**
Bank Street Writer	**87.95**

To order: Send money order, certified cheque, personal cheques must clear our bank, VISA or MASTERCARD. (Include card # and expiry date & signature) Add 5% for shipping and handling. Minimum $3.00 per order. Quebec residents add 9% P.S.T.

INTERNATIONAL MARKETING SERVICES

P.O. Box 522, Boucherville, Quebec, J4B 6Y2
(514) 655-9232

C 64 PROVINCIAL PAYROLL

A complete Canadian Payroll System for Small Business.

• 50 Employees per disk (1541) • Calculate and Print Journals • Print Cheques • Calculate submissions summary for Revenue Canada • Accumulates data and prints T-4s • Also available for 4032 and 8032 Commodore Computers.

Available from your Commodore Dealer.

Distributed by:

MICROCOMPUTER SOLUTIONS

1262 DON MILLS RD. STE. 4
DON MILLS, ONTARIO M3B 2W7
TEL: (416) 447-4811

Disk Software for the Commodore 64™

JOT-A-WORD™

A computerized version of the old five letter word game. Simply pick a secret five letter word (one of the almost 5000 words contained on the disk) and then play against the Jot-A-Word Genie or simply play a solitaire version. Start by typing in a five letter word. The Genie responds by telling you how many letters your guess and the secret work have in common. Don't try to cheat, because the Genie is too smart and it will not accept non-words or continue a game that you have given it wrong scores. This is a simple but stimulating game for ages 9 to senior citizen. A real challenge to your intellect, reasoning powers, logic and deduction skills. It's simply hard to beat; as a fun and educational experience! Graphics and music add to the enjoyment.

ONLY $29.95

Micro-W. D.I.
1342B RT. 23
BUTLER, N.J. 07405

Dealers & Distributors Inquiries Invited **201-838-9027**

Prices are in US dollars.

"The Genie is hard to beat!"

PRO GOLF

Here is your chance to play golf on a championship course without all the headaches of getting a tee time, waiting for that slow foursome ahead of you, losing balls, getting rained out or spoiling a good handicap. This game may be played in the privacy of your home or in a clubhouse lounge for the enjoyment of many members. A challenge to even the best players, this game requires a high degree of practice, expertise and accuracy to attain a good score.

PRO GOLF Features :

- A full range of golf clubs (driveway, fairway wood, wedge and irons 2–9)
- Realistic shot distances depending on club and swing
- The ability to hook or slice a shot
- Up to 4 players in one game
- Detailed, colourful screen layouts of 18 different holes (tee, trees, sandtraps, rough, water, out of bounds)
- Simulated ball reaction to course hazards (e.g. ball bounces off trees)
- Hole distances, par, yards to green, strokes taken on hole, total strokes per round and player totals displayed
- A full screen enlargement of greens for putting
- Accurate putting simulation for angle and distance
- Practice of real golf skills – club selection, type of shot (normal, hook, slice), length of swing, special shot strategy (e.g. chipping, getting around or over trees, water, sandtraps)

PRO GOLF
For The Commodore 64™
$34.95
(diskette only)

written by George Adams
available from your local retailer
distributed by
PACO Electronics Ltd.
20 Steelcase Rd. W.
Markham, ON.
L3R 1B2
416-475-0740

Dealer Inquiries Invited

Name _____

Address _____

Prov/State _____ Postal/Zip Code _____

☐Money Order ☐VISA ☐MasterCard ☐Cheque

Acc# _____ Expiry _____

Please include numbers above name

Add $2.00 for shipping & handling
Ontario residents add 7% sales tax.

WE'LL BACK YOU UP!

ATTENTION COMMODORE 64 OWNERS

If you own a disk drive then you'll need "The Clone Machine". Take control of your 1541 drive.

NEW IMPROVED WITH UNGUARD.*

Package includes:

1.) Complete and thorough users manual
2.) Copy with one or two drives
3.) Investigate and back-up many "PROTECTED" disks
4.) Copy all file types including relative types
5.) Edit and view track/block in Hex or ASCII
6.) Display full contents of directory and print
7.) Change program names, add delete files with single keystroke
8.) Easy disk initialization
9.) Supports up to four drives

MasterCard VISA

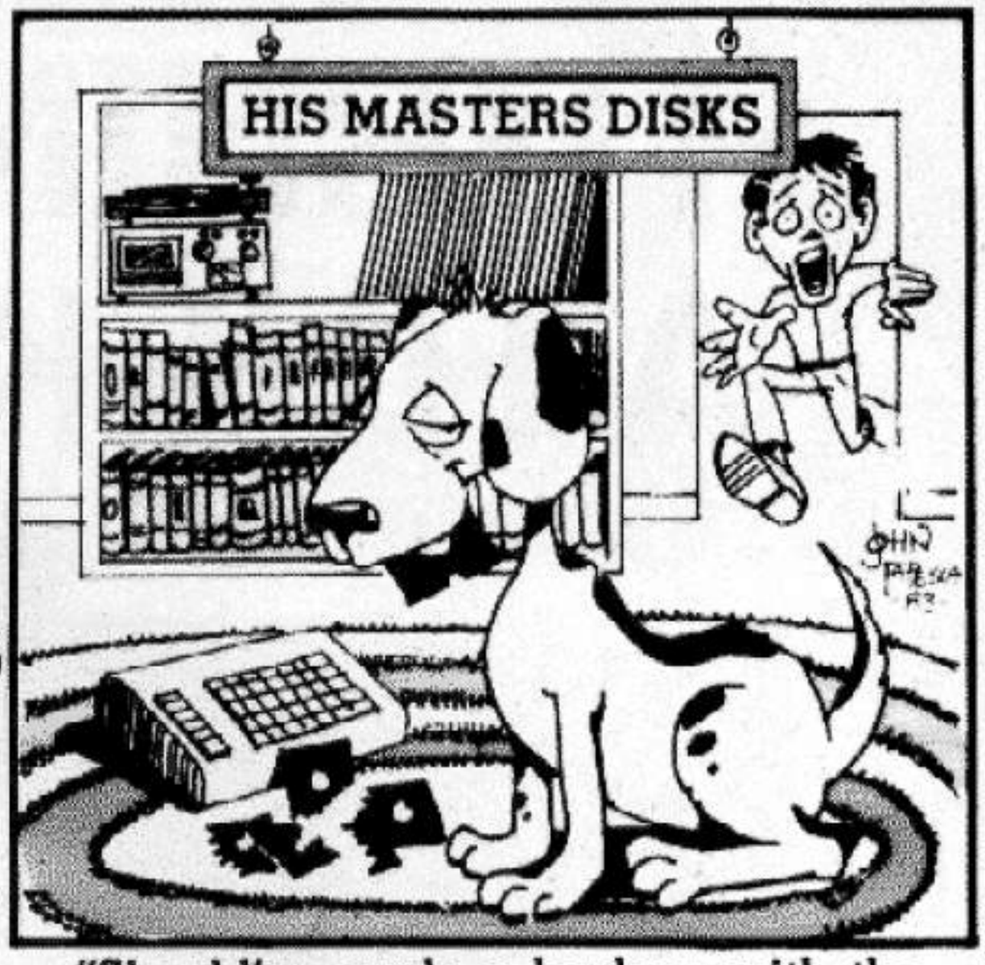
$49.95

*UNGUARD Now allows you to read, write and verify bad sectors and errors on your disk making it easy to back-up most protected software.

Dealers & Distributors
Inquiries Invited

CALL (201) 838-9027

Micro-W. D.I.
1342 B Rt. 23
Butler, N.J. 07405



"Should've made a back-up with the Clone Machine."

COMMODORE COMPUTER PRINTER ADAPTERS

—addressable-switch selectable upper/lower, lower/upper case.
—works with BASIC, WORDPRO, VISICALC and other software.
—IEEE card edge connector for connecting disks and other peripherals to the PET.
—power from printer unless otherwise noted.

RS-232 SERIAL ADAPTER —
baud rates to 9600 — power supply included.
MODEL ADA 1450a $149.00

CENTRONICS/NEC PARALLEL ADAPTER — Centronics 36 pin ribbon connector — handles graphics.
MODEL ADA 1800 . . . . . $129.00

COMMUNICATIONS ADAPTER- serial & parallel ports — true ASCII conversion — baud rates to 9600 — half or full duplex — X-ON, X-OFF - selectable carriage return delay — 32 character buffer — centronics compatible.
MODEL SADI $295.00

COMMODORE 64 to RS-232 CABLE ADAPTER
MODEL ADA 6410 . . . . . $79.00

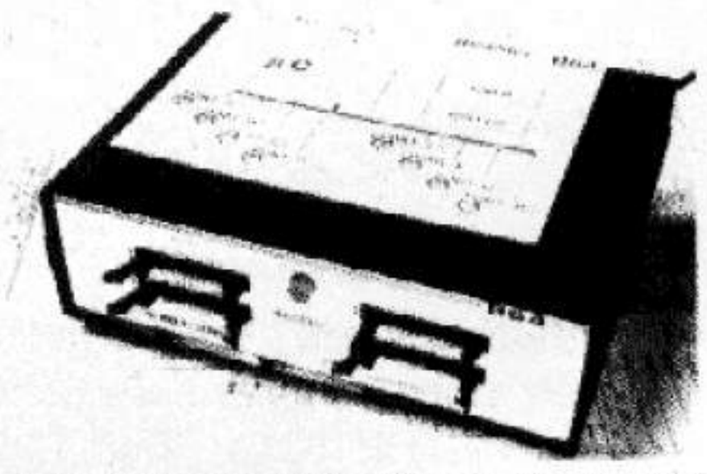
Prices are in US dollars.

BUSSter

COMPUTER INTERFACES

IEEE — 488
RS — 232

ANALOG AND DIGITAL INPUT/OUTPUT MODULES



The BUSSter line of analog and digital products was designed to collect data and to output signals to laboratory and industrial equipment in conjunction with a microcomputer system. These powerful self-contained modules reduce a computer's workload by providing read or write operations to external devices. They are controlled as slave interfaces to real-world physical applications. Control is over an IEEE-488 (GPIB) bus or RS-232 port. BUSSter modules are available in several digital and analog configurations. The internal buffer and timer provide flexibility by allowing the BUSSter to collect data while the host computer is busy with other tasks.

BUSSter A64—64 channel digital input module to read 64 digital signals. Built-in buffer **$495.00**

BUSSter B64—64 channel digital output module to send 64 digital signals **$495.00**

BUSSter C64—64 channel digital input/output module to read 32 and write 32 digital signals. Built-in buffer **$495.00**

BUSSter D16—16 channel analog input module to read up to 16 analog signals with 8 bit resolution (¼%) Built-in buffer **$495.00**

BUSSter D32—32 channel version of the D16 **$595.00**

BUSSter E4—4 channel analog output module to send 4 analog signals with 12 bit resolution (.06%) **$495.00**

BUSSter E8—8 channel version of the E4 **$595.00**

BUSSter E16—16 channel version of the E4 **$695.00**

Add the suffix -G for IEEE-488 (GPIB) or -R for RS-232.

All prices are USA only. Prices and specifications subject to change without notice.

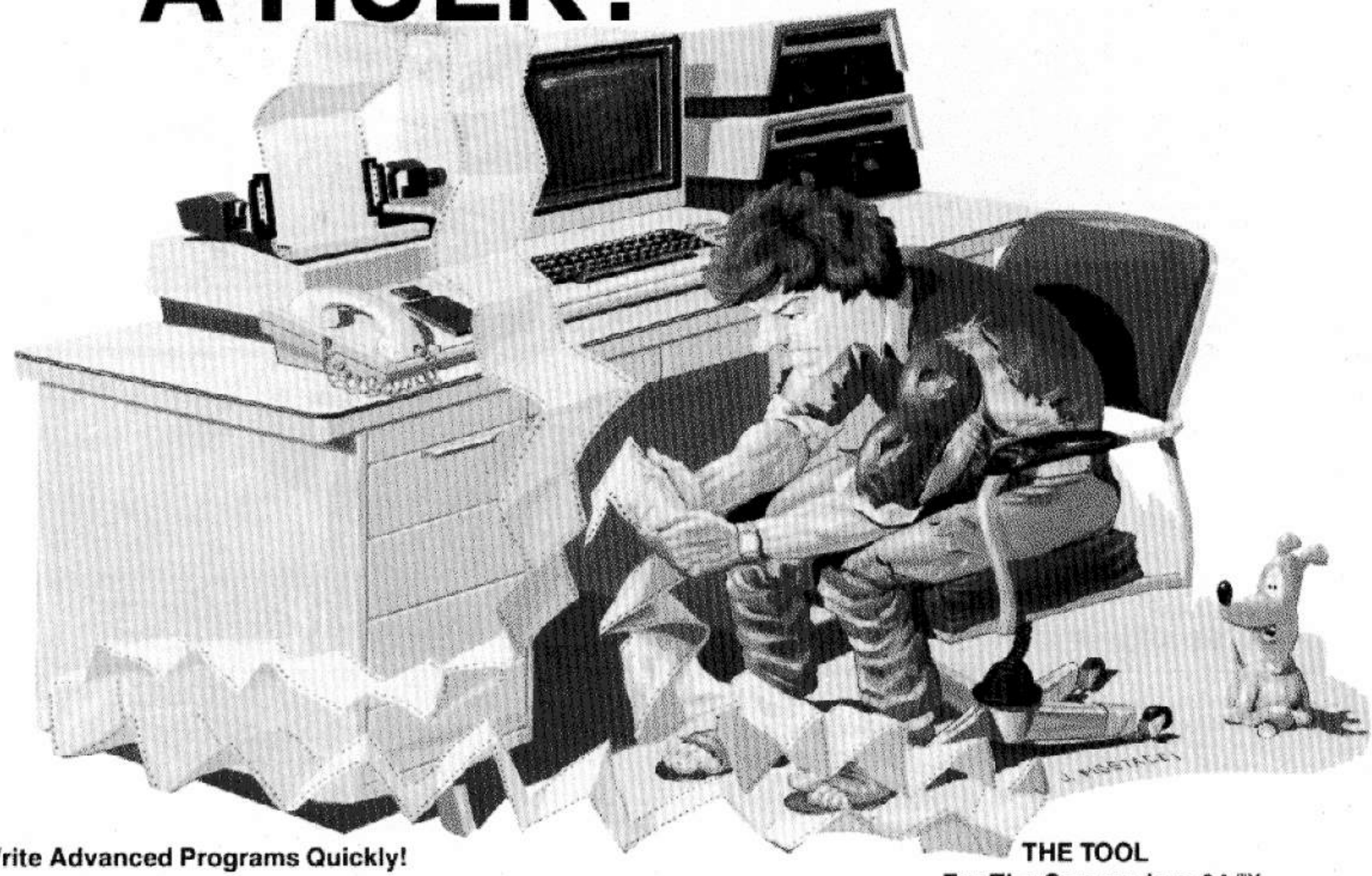
30 DAY TRIAL—
Purchase a BUSSter product, use it, and if you are not completely satisfied, return it within 30 days and receive a full refund.

US Dollars Quoted
$10.00 Shipping & Handling
MASTERCARD/VISA

CmC

Connecticut microComputer, Inc.
INSTRUMENT DIVISION
36 Del Mar Drive
Brookfield, Ct. 06804
(203) 775-4595 TWX. 710-456-0052

IS PROGRAMMING TURNING YOU INTO A HULK?



Write Advanced Programs Quickly!

Tired of writing reams of code? Take a quantum jump into the future! Tommorrow's programmers are using software development tools such as THE TOOL. THE TOOL lets you make use of powerful machine language subroutines. Your programs will execute fast using less code. Input/output routines and professional looking screens are easily created.

Features of **THE TOOL** include :

- Screen Design functions which allow controlled input and output
- High Resolution Graphics with alpha/numeric display
- Screen Save and Load functions (for hi-res and text screens)
- Structured BASIC instructions , e.g. IF THEN ELSE
- Programming Aids (e.g. auto, renumber, delete, find, trace, hardcopy)
- 2 keystroke disk commands (DOS support extensions)
- Game Design Instructions (joy, scroll, screen, colour)
- A 50 page user manual

THE TOOL
For The Commodore 64 ™
$65.00
(diskette only)

developed by Micro Application
available from your local retailer
distributed by
PACO Electronics Ltd.
20 Steelcase Rd. W.
Markham, ON.
L3R 1B2
416-475-0740

Dealer Inquiries Invited

Name _____

Address _____

Prov/State _____ Postal/Zip Code _____

Order ☐VISA ☐MasterCard ☐Cheque

Acc# _____ Expiry _____

Please include numbers above name

Add $2.00 for shipping & handling
Ontario residents add 7% sales tax.

CANADIAN SOFTWARE SOURCE

COMPUTERWARE CATALOGUE SHOPPING

GREAT SAVINGS
FAST*
FRIENDLY
SERVICE

► VISIT US AT THE PERSONAL AND BUSINESS COMPUTER SHOWPLACE, APRIL 6 to 8/84 C.N.E. TORONTO

OVER 700 COMMODORE 64 PRODUCTS

CANADIAN DOLLARS	Retail	C.S.S.		Retail	C.S.S.
HOME ACCOUNTANT (Continental) (D)	$ 99.95	$ 74.95	WEIGHT CONTROL (Simutech) (CAR)	$ 69.95	$ 52.95
BANK STREET (BRODERBUND) (D)	99.95	69.95	BEAR JAM (Chalkboard) (CAR)	49.95	42.95
PAINTMAGIC (Datamost) (C & D)	69.95	56.95	EARLY GAMES FOR YOUNG CHILDREN (Cntpt.) (T & D)	39.95	34.95
COMBAT LEADER (SSI) (D)	49.95	44.95	COTCHA MATH GAMES (Comm *Data) (D)	Call	Call
KOALA PAD (Koala Tech)	149.95	107.50	MATH TUTOR (Comm *Data) (D)	Call	Call
KEN USTON'S BLACKJACK (Screenplay) (D)	94.95	82.50	SPELLCOPTER (Designware) (D)	49.95	44.95
MINER 2049'er (RESTON) (CAR)	59.95	49.95	TRIVIA 1 (Cymbal) (D)	49.95	44.95
STARFIRE/FIRE ONE (Epyx) (T & D)	49.95	39.95	SYSRES (Solidus) (D)	115.95	99.95
WAY OUT (Sirius) (D)	51.95	45.95	PRINTER INTERFACE +G (Cardco)	149.95	119.95
FAMILY TREE GENEALOGY (D)	49.95	46.95	GEMINI X (Star Micronics) including PRINTER INTERFACE +G (Cardco)	Special	495.00
FCM (Continental)	69.95	54.95	BEACH HEAD (ACCESS) (T & D)	69.95	37.95
THE HYPNOTIST (Psycom) (D)	149.95	139.95	IN SEARCH OF (Spinnaker) (D)	52.95	42.95
FLIGHT SIMULATOR II (Sublogic) (D)	59.95	50.95	PRINTER PAPER 3OM 9½" x 11" (3300 shts box)	Special	42.95
STRIP POKER (Artworx) (D)	54.95	48.95	MUSIC CALC (Wave form) (D)	100.00	75.00
ASTROPOSITIONS (King Microware) (D)	49.95	43.95	MUSIC CONSTRUCTION SET (Electronic Arts) (D)	59.95	53.95
PAPER CLIP (Batteries included)	**IBC	99.95	JOYSTICK (Kraft)	Special	25.00
FRENCH EASYSCRIPT 64 (Silicom Int.) (D)	**IBC	129.95			
SEXUAL CONFIDENCE (Simutech) (CAR)	69.95	58.95			

*IBC — INCLUDES 10% BONUS CREDIT

FOR ORDER OR FREE CATALOGUE WRITE OR PHONE CANADIAN SOFTWARE SOURCE

BOX "340" STATION "W", TORONTO, ONTARIO M6M 5B9 (416) 491-2942

VISA

Ontario Residents add 7% sales tax. Send certified cheque or money order. Visa and Master Card please include card number, expiry date and signature.

*Delivery by U.P.S. within 3 days of order date if stocked by local suppliers.

Add $2.50 for shipping and handling. All items subject to availability. Prices subject to change without notice.

MasterCard

COMMODORE OWNERS

Join the world's largest, active Commodore Owners Association.

- Access to thousands of public domain programs on tape and disk for your Commodore 64, VIC 20 and PET/CBM.
- Monthly Club Magazine
- Annual Convention
- Member Bulletin Board
- Local Chapter Meetings

Send $1.00 for Program Information Catalogue.
(Free with membership).

Membership Fees for 12 Months	Canada	— $20 Can.
	U.S.A.	— $20 U.S.
	Overseas	— $30 U.S.

T.P.U.G. Inc.
Department "M"
1912A Avenue Road, Suite 1
Toronto, Ontario, Canada M5M 4A1

* LET US KNOW WHICH MACHINE YOU USE *

COMMODORE 64™ COMAL ADDS:

- 40 Graphics Statements
- 10 Sprite Statements
- "LOGO" TURTLE GRAPHICS
- RUN-TIME COMPILER
- FAST program execution
- auto line numbering
- line renumbering
- program structures
- merging program segments
- long variable names
- named procedures
- parameter passing
- local and global variables
- random access disk files
- stop key disable
- End Of File detection

What does this and more? **COMAL**
What is the cost? **Only $19.95**

All this and much, much more on disk with many sample programs. ONLY $19.95. Also available: COMAL HANDBOOK, $18.95. BEGINNING COMAL, $19.95. STRUCTURED PROGRAMMING WITH COMAL, $24.95. FOUNDATIONS IN COMPUTER STUDIES WITH COMAL, $19.95. CAPTAIN COMAL GETS ORGANIZED, $19.95. COMAL TODAY newsletter, $14.95. Send check or Money Order in US Dollars plus $2 handling to: COMAL Users Group, U.S.A., Limited. 5501 Groveland Ter., Madison, WI 53716 phone: 608-222-4432. COMMODORE 64 is trademark of Commodore Electronics Ltd. CAPTAIN COMAL is trademark of COMAL Users Group, U.S.A., Limited.

Advertising Index

Software

Advertiser	Issue# / Page 03	04	05	06	01	02	Product Name (Description)	Manufacturer
Academy Software		73	77	78	89	**98**	VIC20/C64 Software	
Bennett Software Co.						**92**	Graph-Term 64	
Boston Educational Computing		64					Educational Software	
Canadian Software Source				79	91	**103**	C64 Software	
Cass-A-Tapes					88		Commodore software	
Cardinal Software			72	62	83	**97**	VIC20/C64 Games, Utilities, Edu.	
COMAL Users Group				65	88	**103**	C64 COMAL	
Computer Alliance						**94**	BASM (language for the 64)	
Dexterity Software					82		C64/VIC 20 games	
Dyadic Resources Corp.					88		SuperPET information	
Eastern House		68	81	71			MAE Assembler	
Info Mag Inc.					79	**95**	Commodore software	
Input Systems Inc.		68	81	71			Typro (wordprocessor)	
Isis Hathor			IBC	IBC	IBC	**IBC**	Laser Strike	
King Microware				76	87	**2**	VIC/64/PET software	
Magreeable Software		66	75	74	81	**94**	Stock Helper	
Microcomputer Solutions			81	71	86	**99**	C64 Provincial Payroll	
Micro-Fax						**95**	C64 Software	
MicroSpec			70	69	91		C64/VIC 20 Business Software	
Microtechnic Solutions			77	78	89	**98**	C64 Terminal software	
Micro W.D.I.			75	74	81	**99**	C64 JOT-A-WORD	
			82	65	82	**101**	C64 Disk Utility	
						96	Mr. Tester (diagnostic prog.)	
						96	Fantastic Filer	
Midwest Micro Inc.		69					VIC20/C64 Graphics Util	
			83	73	80	**97**	VIC20/C64 SuperTerm	
PACO Electronics Ltd.			73	72	90	**102**	The TOOL (programming aid)	Micro Application
			76	68	84	**100**	Pro Golf	
Performance Micro Products		64		65			C64 Forth	
P.F. Communications			86				J Butterfield video tutor	
Pro-Line Software	79	73	72	62	2	**IFC**	Commodore software	
	79	73	72	62	83		PAL 64 (assembler)	
	76	69	74	77	85		POWER 64 (programming aid)	
		74	78	64	86		MailPro	
	76	66	83	73	80	**92**	general	
Psycom Software Int'l			80	79			C64 software	
Public Domain Inc.					78	**97**	Commodore software	
Silicom International				75			SuperBase 64 (data base)	Precision Software
William Robbins Software		74	75	74	81	**93**	VIC/64/PET Software	

Hardware

Advertiser	Issue# / Page 03	04	05	06	01	02	Product Name (Description)	Manufacturer
Apropos Technology			79	67			VIC20/C64 Printer, Exp board	
cgrs Microtech						**94**	A/D Converter, Quikdisk	
Connecticut microComputer		64	78	64	86	**101**	Analog/Digital I/O	
Eastern House		68					Trap 65	
		68					VIC Rabbit	
		68					Eprom Programmer	
		68					Communications Bd	
George M. Drake & Associates		BC	BC	BC	BC	**BC**	Colour Monitors	Amdek
Micro W.D.I.		70					Tape Interface	
		70					VIC20 RAM Expand	
Micro World Electronix		65	74	77	85		VIC20/C64 Printer Interface	
Midwest Micro Inc.		69				**97**	Smart ASCII Plus	
Midwest Peripherals		74	72	62	83		VIC20 Expander	
Precision Technology	79	73					VIC20/C64 Expander Boards	
Richvale Telecommunications	IFC	IFC	IFC	IFC	IFC		C64 Link (IEEE adapter) + software	
Zanim Systems			84	70			Home control hardware	

Accessories

Advertiser	Issue# / Page 03	04	05	06	01	02	Product Name (Description)	Manufacturer
Bytes & Pieces Inc.						**93**	C64 'Cheat Sheets'	
The Book Company					91		Software review/exchange	
The Code Works		69	78	64			'CURSOR', C64 Tape Magazine	
Hunter Nichol						**93**	World Of Commodore II show	
Int'l Marketing Services			80	79	79	**99**	Disk, printers, misc.	
Midnight Software Gazette		72	71	66		**95**	Subscriber Info	
Toronto PET Users Group		75	74	77	85	**103**	Membership info	
Zanim Systems					1	**1**	CAD/CAM Tutorial	

Laser Strike

for the commodore 64

challenge the asteroid field,
maneuver the caves of ice,
experience the thrill,
play laser strike.

Laser strike, written in full machine language for the Commodore 64.

Commodore 64 is a registered trademark of Commodore Business Machines Inc.

Visa/MC/Check/Money Order accepted

In U.S.
Cassette $24.95
Disk $29.95
Isis Hathor Digital Productions
6184 Verdura Ave.
Goleta, CA 93117
(805) 964-6335
Add $2.00 postage and handling
California residents add 6% sales tax

*** Ask about Laser strike posters**

ISIS HATHOR
DIGITAL PRODUCTIONS

In U.K.
Cassette £ 9.00 VAT included
Disk £19.95 VAT included
Isis Hathor U.K.
Andrew Barrow
Royden, Perkslane
Prestwood, Gt. Missenden
Bucks, England HP16 OJD
02406-3224
You will be billed for postage and handling

COMPATIBLE COLOR-I . . .

NEW 2 YEAR WARRANTY!
On all monitor electronics . . . 3 yrs. on all CRT's
(See details at dealer)

The popular choice for popular computers . . . at a popular price.

The Color-I Monitor is designed to perform superbly with your Apple II, Atari or VIC Commodore personal computer and others. Highly styled cabinet. It accepts a composite video signal to produce vivid, richly colored graphic and sharp text displays. Very reasonably priced, the Color-I is a giant step above home TV sets and other monitors.

Just write, or call to receive complete specifications on the Amdek Color-I Monitor.

- Quality 260(H) x 300(V) line resolution.
- Built-in speaker and audio amplifier.
- Front mounted controls for easy adjustment.
- Interface cables available for Atari and VIC Commodore computers.
- FCC/UL approved.

2201 Lively Blvd. • Elk Grove Village, IL 60007
(312) 364-1180 TLX: 25-4786
REGIONAL OFFICES: Calif. (714) 662-3949 • Texas (817) 498-2334

AMDEK CORP.

Amdek . . . your guide to innovative computing!